Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Automatic Scaling in Cloud Computing

Doctoral Thesis

by

Ing. Tomáš Vondra

Prague, January 2017

Ph.D. programme: Electrical Engineering and Information Technology
Branch of Study: Artificial Intelligence and Biocybernetics

Supervisor: Ing. Jan Šedivý, CSc.

Abstract and contributions

This dissertation thesis deals with automatic scaling in cloud computing, mainly focusing on the performance of interactive workloads, that is web servers and services, running in an elastic cloud environment. In the first part of the thesis, the possibility of forecasting the daily curve of workload is evaluated using long-range seasonal techniques of statistical time series analysis. The accuracy is high enough to enable either green computing or filling the unused capacity with batch jobs, hence the need for long-range forecasts. The second part focuses on simulations of automatic scaling, which is necessary for the interactive workload to actually free up space when it is not being utilized at peak capacity. Cloud users are mostly scared of letting a machine control their servers, which is why realistic simulations are needed. We have explored two methods, event-driven simulation and queue-theoretic models. During work on the first, we have extended the widely-used CloudSim simulation package to be able to dynamically scale the simulation setup at run time and have corrected its engine using knowledge from queueing theory. Our own simulator then relies solely on theoretical models, making it much more precise and much faster than the more general CloudSim. The tools from the two parts together constitute the theoretical foundation which, once implemented in practice, can help leverage cloud technology to actually increase the efficiency of data center hardware.

In particular, the main contributions of the dissertation thesis are as follows:

1. New methodology for forecasting time series of web server load and its validation
2. Extension of the often-used simulator CloudSim for interactive load and increasing the accuracy of its output
3. Design and implementation of a fast and accurate simulator of automatic scaling using queueing theory

Keywords:
cloud computing, autoscaling, time series forecasting, green computing, simulation

Abstrakt a přínos práce

Tato dizertační práce se zabývá cloud computingem, konkrétně se zaměřuje na výkon interaktivní zátěže, například webových serverů a služeb, které běží v elastickém cloudovém prostředí. V první části práce je zhodnocena možnost předpovídání denní křivky zátěže pomocí metod statistické analýzy časových řad se sezónním prvkem a dlouhým dosahem. Přesnost je dostatečně vysoká, aby umožnila buď šetření energií nebo vyplňování nevyužité kapacity dávkovými úlohami, jejichž doba běhu je hlavním důvodem pro potřebu dlouhodobé předpovědi. Druhá část se zaměřuje na simulace automatického škálování, které je nutné, aby interaktivní zátěž skutečně uvolnila prostor, pokud není vytěžována na plnou kapacitu. Uživatelé cloudů se převážně bojí nechat stroj, aby ovládal jejich servery, a právě proto jsou potřeba realistické simulace. Prozkoumali jsme dvě metody, konkrétně simulaci s proměnným časovým krokem řízeným událostmi a modely z teorie hromadné obsluhy. Během práce na první z těchto metod jsme rozšířili široce používaný simulační balík CloudSim o možnost dynamicky škálovat simulovaný systém za běhu a opravili jsme jeho jádro za pomoci znalostí z teorie hromadné obsluhy. Náš vlastní simulátor se pak spoléhá pouze na teoretické modely, což ho činí přesnějším a mnohem rychlejším nežli obecnější CloudSim. Nástroje z obou částí práce tvoří dohromady teoretický základ, který, pokud bude implementován v praxi, pomůže využít technologii cloudu tak, aby se skutečně zvýšila efektivita využití hardwaru datových center.

Hlavní přínosy této dizertační práce jsou následující:

1. Stanovení metodologie pro předpovídání časových řad zátěže webových serverů a její validace
2. Rozšíření často citovaného simulátoru CloudSim o možnost simulace interaktivní zátěže a zpřesnění jeho výsledků
3. Návrh a implementace rychlého a přesného simulátoru automatického škálování využívajícího teorii hromadné obsluhy

Klíčová slova:

cloud computing, automatické škálování, předpovídání časových řad, úsporné výpočty, simulace

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Goals of the Dissertation Thesis . . . 3
- 1.4 Theoretical Background . . . 4
  - 1.4.1 Private Cloud . . . 4
  - 1.4.2 Public Cloud . . . 6
  - 1.4.3 Technical limitations . . . 7
  - 1.4.4 Capacity Planning . . . 8
  - 1.4.5 Automatic scaling . . . 10
  - 1.4.6 Predictive scaling . . . 11
  - 1.4.7 Evaluation and Metrics . . . 12
  - 1.4.8 Queueing Theory . . . 13
  - 1.4.9 Queueing Networks . . . 16
- 1.5 Structure of the Dissertation Thesis . . . 19

2 State-of-the-Art and Related Work **21**
- 2.1 Performance Prediction . . . 21
  - 2.1.1 Research topics . . . 21
  - 2.1.2 Employed methods . . . 26
  - 2.1.3 Summary . . . 33
- 2.2 Autoscaling Simulation . . . 34
  - 2.2.1 CloudSim and other cloud simulators . . . 34
  - 2.2.2 Analytical solvers for queueing network and other formal models . . . 35
  - 2.2.3 Languages for discrete event simulation . . . 38

3 Overview of Our Approach **41**
- 3.1 Interactive Workload Prediction . . . 41
  - 3.1.1 Holt-Winters exponential smoothing . . . 42

3.1.2 Box-Jenkins / ARIMA models . . . 43
3.2 Simulation of Cloud Autoscaling . . . 44
3.2.1 CloudSim and Changes to implement Autoscaling . . . 45
3.2.2 Verification of CloudSim using Load Testing . . . 47
3.3 Custom Simulator Design based on Queueing Theory . . . 51

4 Main Results **57**
4.1 Feasibility of Interactive Workload Prediction . . . 57
4.1.1 Loading of data . . . 57
4.1.2 Time series diagnostics . . . 58
4.1.3 Holt-Winters model fitting and evaluation . . . 60
4.1.4 ARIMA model selection . . . 62
4.1.5 Model validation . . . 66
4.1.6 Comparison of the two model families . . . 67
4.1.7 Forecast plots . . . 69
4.2 CloudSim Modifications for Interactive Traffic . . . 71
4.2.1 Autoscaler implementation in CloudSim . . . 71
4.2.2 Modifications to increase accuracy . . . 77
4.3 Cloud Simulator based on Queueing Theory . . . 80
4.3.1 Definitions of data and metrics . . . 80
4.3.2 The utilization-based autoscaler . . . 83
4.3.3 The latency-based autoscaler . . . 85
4.3.4 The queue length-based autoscaler . . . 87
4.3.5 Second threshold for overload detection . . . 89
4.3.6 The latency-queue hybrid autoscaler . . . 90
4.3.7 The latency-utilization hybrid autoscaler . . . 91
4.3.8 Summary of results . . . 93
4.3.9 Stability of the hybrid algorithms with increased load . . . 94
4.3.10 Simulation plots . . . 95
4.4 Minor results and developed software . . . 100
4.4.1 Cloud Gunther . . . 100
4.4.2 ScaleGuru . . . 101
4.4.3 Private IaaS cloud comparison . . . 102
4.4.4 Private PaaS cloud comparison . . . 102
4.4.5 Scheduling algorithms in job queues . . . 103
4.4.6 Data movement in Hybrid Clouds . . . 103
4.4.7 Automatic cloud deployment driven by a performance model . . . 103
4.4.8 Private cloud monitoring . . . 104
4.4.9 Autoscaling in Cloud Foundry . . . 104

5 Conclusions **107**
5.1 Workload Forecasting . . . 107
5.2 CloudSim Changes . . . 108

5.3 Custom simulator design 109
5.4 Summary 111
5.5 Contributions of the Dissertation Thesis 111
5.6 Future Work 111

Bibliography **113**

A List of Publications and Grants **123**
List of candidate's work related to the thesis 123
List of candidate's work non-related to the thesis 125

List of Figures

1.1 Daily CPU load plot of a web server . . . 9
1.2 Amount of server instances serving a website during a day . . . 9
1.3 Diagram of a queueing system . . . 14
1.4 Throughput-latency eye diagram . . . 17
1.5 A simple queueing network describing a database system . . . 17

3.1 The constant incoming load at the load balancer . . . 49
3.2 Load as seen by the last web server (the one not shut down) . . . 49
3.3 Latency at the Tsung load generator, last step average highlighted . . . 49
3.4 Latency results from CloudSim . . . 50
3.5 Concurrency at the load balancer . . . 50
3.6 Concurrency at the last web server (capped at 20 to prevent swapping) . . . 51
3.7 Result from CloudSim plotted against load . . . 51
3.8 Results from PDQ in R . . . 52
3.9 Flowchart of the autoscaling simulator . . . 53

4.1 oe series decomposition, from top to bottom: overall time plot, trend, seasonal and random compoment . . . 59
4.2 gaff series seasonal subseries plot . . . 59
4.3 Autocorrelogram of residuals of the H-W model on bender . . . 62
4.4 ACF of oe without and with differencing . . . 64
4.5 Class diagram of CloudAnalyst [1] . . . 72
4.6 Class diagram of CloudSim [2] . . . 73
4.7 Simulated latency when adding VMs while keeping load constant . . . 75
4.8 Simulated daily load curve, amount in requests per hour . . . 76
4.9 Simulated latency with a constant number of VMs . . . 76
4.10 Simulated latency with automatic scaling . . . 77
4.11 Simulation results from modified CloudSim . . . 79
4.12 Result from modified CloudSim plotted against load . . . 79

4.13 Time series oe and the latency-queue hybrid autoscaler with settings 60 ms up and 1 req down. Six figure plot mode. 96
4.14 Time series bender and the classic latency-based autoscaler with settings 300 ms up and 152 ms down. Five figure plot mode. 97
4.15 Time series bender and the classic latency-based autoscaler with settings 300 ms up and 152 ms down. Apdex component plot. 98
4.16 Time series gaff and the latency-utilization hybrid autoscaler with parameters 130 ms up and 10% down with overload detection. Five figure plot mode. . . . 99

List of Tables

4.1 Evaluation of the Holt-Winters model on out-of-sample data 61
4.2 Order of differencing based on unit root tests 63
4.3 Last significant lags and model orders 65
4.4 Parameters of the estimated ARIMA models and their validation measures . . 67
4.5 Evaluation of the ARIMA models on out-of-sample data 68
4.6 Simulation and Load Test Results 80
4.7 Utilization-based autoscalers on “oe” 84
4.8 Utilization-based autoscalers on “bender” 84
4.9 Latency based autoscalers on “oe” 85
4.10 Latency-based autoscalers on “bender” 85
4.11 Utilization-based autoscalers on “gaff” 86
4.12 Latency-based autoscalers on “gaff” 86
4.13 Queue length-based autoscalers on “oe” 88
4.14 Queue length-based autoscalers on “bender” 88
4.15 Queue length-based autoscalers on “gaff” 89
4.16 Queue length-based autoscalers with overload detection on “gaff” 89
4.17 Latency-Queue hybrid autoscalers on “oe” 91
4.18 Latency-Queue hybrid autoscalers on “bender” 91
4.19 Latency-Utilization hybrid autoscalers on “oe” 92
4.20 Latency-Utilization hybrid autoscalers on “bender” 92
4.21 Latency-Queue hybrid autoscalers with overload detection on “gaff” 93
4.22 Latency-Utilization hybrid autoscalers with overload detection on “gaff” . . . 93
4.23 Summary table 94

Chapter 1

Introduction

1.1 Motivation

The Cloud is becoming ubiquitous. It is considered as a deployment option with nearly any software project nowadays. Nevertheless, the workloads on the cloud are very often legacy static applications.

The properties of the Cloud, mainly the elasticity in the amount of resources used by an application and the agility in the reuse of physical resources between different applications and even different users, which were not present in server virtualization products before the cloud age, can only be used if applications are engineered with them in mind.

Elasticity can be exploited only if there is an autoscaling element in the applications, which monitors their resource usage and changes the amount of used resources accordingly. This element is also known as the self-adaptive load balancer. From our experience, application developers and deployers do not have enough knowledge of cloud performance engineering and are likely to commit too much resources rather than risk wrong autoscaler settings. In this work, we present a way to simulate the autoscaler with different algorithms and settings, which should help alleviate fears associated with its use and help with estimation of cloud service costs.

If there is an autoscaler on every interactive workload in a cloud (we are mostly concerned with Private Cloud clusters), resources will get properly released in periods of low usage. This will allow the provider to shut down inactive compute resources, which is a key element of Green Computing, which we define as a drive towards consolidating workload to as few machines as possible while maintaining SLA (Service Level Agreements).

Green computing is a part of the provider's view on the cloud. Embracing the concept of IaaS (Infrastructure as a Service), i.e. offering computing power as a service between subjects (be it two companies or just departments inside the same one), optimization can be done either on the side of the provider or the customer. The provider of cloud services wants to minimize the running costs of the system while maintaining GoS (Grade of Service) for the customer, which at the level of IaaS means availability of memory, CPU cycles and network and disk bandwidth in agreed upon amount and quality.

The running costs can be best minimized by turning off excess physical computers (frequency scaling of CPUs is more agile, but has lower power impact, see Section 1.4, Subsection 1.4.6. To maintain GoS and avoid unnecessary on/off cycles, load prediction can be implemented at the level of the data center.

Leaving compute resources active even in periods of low loads enables a data center to achieve good values of PUE (Power Usage Effectiveness), which is a measure of data center efficiency comparing the total power draw to the power draw of only the computing components (mainly excluding cooling and power supply power usage). When servers are shut down at night, this indicator will become worse, but the total energy usage should be lower, which is what matters the most.

In our article [3], we proposed to exploit cloud agility between interactive and batch computations. To do this, a service which forecasts the usage of resources of the interactive tasks would be needed, so that long-running grid-type tasks could be run without the risk of being terminated because the interactive workloads need more computing power.

The forecasting service would use data about resources used on the cluster by interactive traffic (taken from the autoscalers), do a prediction on that, and fill the unused resources by batch jobs, yielding a cluster that is highly utilized all the time, but with minimal job preemptions or terminations (depending on the batch queue used). In this work, we demonstrate the feasibility of forecasting the load of several different purpose web services using statistical methods. The result can be utilized not only on the provider's side to turn computers off at night or schedule batch jobs in periods of low activity but also on the client's side for predictive autoscaling, decreasing their cloud service costs. It answers the question: "How many slots for VMs will be used for the next X hours with probability P?"

The proposed forecasting service was not implemented yet because of a lack of demand. From conversations with multiple companies, the most desired place to apply optimization is currently the customer side of IaaS. It also correlates with the lower proliferation of private cloud, particularly in the Czech Republic. The companies are mostly clients of public clouds and would like advice in the form of performance models or autoscaler deployment settings that would let them save on cloud costs. Moreover, the client-side optimization can also be employed in a private cloud, once they build it. On the side of the cloud client, who does not see the infrastructure, there is space for optimization inside the autoscaler, which is where the proposed simulation platform comes in. The goal is to minimize the cost the customer pays for cloud resources while maintaining GoS for end users. The end user GoS is defined as the response time distribution of the web application.

We think that performance prediction will also make a significant contribution to autoscaler quality and will be implemented in the simulation framework in the next release, which is, however, out of the scope of this work. Current autoscalers available for both private and public clouds are reactive, meaning that they can add resources after a threshold of some monitored performance variable is breached. If the autoscaler contained some prediction mechanism, it could add resources before the threshold was breached. That would allow the thresholds to be set higher without affecting GoS.

The prediction methods for entire data centers and single applications may not be the

same and may require different parameter settings. However, datacenter-scale data is not easy to obtain. We tried negotiating with Czech companies offering cloud computing, such as TC Pisek, Odorik, Master Internet, Forpsi, and PonyCloud. The problems we encountered were mostly that their systems are too small and are not ready for autoscaling or virtual machine migration. We did not get any response from other European cloud companies except one, which offers Cloud Foundry, which in its open-source version lacks any monitoring functions, which are a prerequisite for autoscaling. With emerging global cloud providers such as Mega and Digital Ocean, there was already a problem with data privacy concerns. Our only source of data remains at the Masaryk University in Brno, which operates the Czech national grid Metacentrum and, as an associated service, a large OpenNebula cluster MetaCloud[1]. The problem with this data is that, in contrast with a business cloud, a scientific cloud lacks any interactive services (only about 1% of the virtual machine traces have daily seasonality, which indicates human interactive use). Therefore, the methods presented in this work are evaluated on data from a static web server farm of a web hosting company. However, the data contains several different purpose and scale servers and both normal and Christmas holiday periods, making it quite interesting.

1.2 Problem Statement

This dissertation thesis studies the ways to either increase the utilization of a private cloud or reduce the operating costs of a client in the public cloud, where the latter is actually a subproblem of the former. The problem is that cloud offerings claim that using them will bring cost reductions over owned hardware with virtualization in the case of the private cloud or IT outsourcing in the case of the public cloud. However, the main advantage of the cloud over the older technologies is the possibility of automation using web services and elasticity with pay-as-you-go billing. In practice, the cloud infrastructure is still being used statically, which prevents the realization of its potential.

1.3 Goals of the Dissertation Thesis

1. Study the nature of interactive cloud workloads to be able to scale them to match demand

2. Study the properties of automatic scaling and simulate autoscaler deployments

3. Propose the strategies to a) increase the utilization of private clouds and b) the scaling settings to reduce the operating costs in the public cloud.

[1] `http://www.metacentrum.cz/en/cloud`

1.4 Theoretical Background

Cloud computing is the last advance in data center management. In its IaaS (Infrastructure as a Service) form, it allows for rapid provisioning of virtualized server, storage and network resources with minimal user interaction. Using add-on configuration and deployment tools, or the next layer of cloud, PaaS (Platform as a Service), applications can be deployed to these server instances. The automation possibilities given by cloud APIs (Application Programming Interfaces) offer lots of ground for research on how to use the resources optimally and for better user satisfaction.

As the term "as a service" suggests, a common property of cloud technologies is the separation of two entities, the provider and the consumer of a service. In IaaS, the commodity is computing power (and also memory, storage space, and network bandwidth) in the form of virtual machines. Therefore, the cloud is really a set of new features over server virtualization. Depending on the relationship between the producer and consumer of the service, we distinguish between the private and the public cloud. When they are used together, we speak of the hybrid cloud.

A public cloud provider offers a generally available service, where users can run a large number of instances, and for all practical purposes, it offers them the illusion of infinite supply. The pay-as-you-go billing model then offers high flexibility – using the cloud does not entail any capital expenditure at all, only operating costs. However, according to our experience, the costs of the major providers are higher than those of owned hardware for long-term usage.

The cloud, private or public, besides being a platform for web applications, can also take the role previously occupied by the grid, that is being a batch computing platform. The upside is that when the demand for computations is not constant, instances can be created for the job and then terminated, along with the associated costs. The downside is that traditional computing tools are not ready for the cloud, but expect a dedicated cluster of machines. Launching them in the cloud is possible, but it wastes resources. New cloud-aware tools are needed. One of them is Cloud Gunther.

1.4.1 Private Cloud

The private cloud is built by the same company that is using it (we are excluding the "virtual private clouds," which are actually a section of a public cloud separated by network virtualization). The advantage of a private cloud over a physical or virtualized data center is the agility of infrastructure. The automation of IaaS offers to create and destroy virtual machines easily, and it can be done algorithmically.

The same features as stated above for the public cloud apply here. However, the capacity of the private cloud is not (even seemingly) infinite and is a known quantity. If it is not sufficient (all of the time or, perhaps, only on peak hours), then the user can turn to hybrid cloud and cloudbursting. Also, whereas the user is not interested in how a public cloud provider uses excess capacity, in a private cloud, the owner may desire to use some capacity also for secondary tasks.

The private cloud infrastructure may be used to run static virtual servers, where each of them is assigned compute resources (CPU and memory) based on its expected peak usage. The cloud operating system then starts it on an available physical machine based on its scheduling algorithm, and it will run there permanently. However, with this approach, the private cloud will probably fail to satisfy expectations regarding resource savings over traditional virtualization.

With server virtualization, the administrator knows, which virtual machines share common hardware and is aware of their performance profiles. Therefore, he or she can assign more resources to the virtual machines, than what the physical machine actually has (overcommitment). In the case of overload, some VMs may be migrated elsewhere.

With the automatic resource assignment present in the cloud, one can also expect savings due to resource sharing, but overcommitment is also automatic, managed by the cloud scheduler, and therefore riskier because we do not know the assignment between virtual and physical machines in advance. While live migration is possible, it is mostly used to hide planned outages from clients, not for performance optimization. That would require the cloud administrators to know the performance profile of the applications, which is not a concern of theirs, but of the application administrators. Automatic workload consolidation tools are, sadly, still in the realm of scientific papers.

However, cloud computing has its advantages. The most important one is the elasticity of compute resources, which is connected with the illusion of infinite supply. An application running in the cloud can elastically increase or decrease the amount of resources assigned to it based on the actual user demand for it. The illusion, of course, ends, when all resources on the cloud cluster are exhausted, but until then, it is a powerful capability. Because the same way as applications share resources on a single machine with static server virtualization, in the cloud, we can imagine application resource allocations flowing over the whole cluster or datacenter, based on their individual resource demands.

This key property allows for significant savings on compute infrastructure, because it is no longer necessary to procure enough servers to cover the resource spikes of each application, even if they are not occurring at the same time, as when each had its dedicated server farm. Rather, the infrastructure is common for all applications, yet it ensures separation of users and applications (multi-tenancy), and it may be dimensioned to suit the sum of resource demands of all the tenants in worst case / at the worst time.

Moreover, if the computing power is not required by any application at some point, the physical servers may shut down automatically to save energy, only to be powered on again, when the demand rises. It is irrelevant if a particular server fails at power-on because they are all identical and do not store valuable data.

The elastic scaling of cloud resources (both up and down) can be performed either by the application administrator by hand, or, better, delegated to an automatic tool.

Because private clouds emerged only after the public offerings, the public clouds offer a much broader functionality and service set than their private counterparts (compare e.g. Amazon Web Services and Eucalyptus, which have a compatible API.). Most importantly, there are not many autoscaling applications for private clouds, one of them being our project ScaleGuru.

1.4.2 Public Cloud

The Public Cloud brings the illusion of infinite supply, meaning that any application may use as much resources of the provider's datacenter as required by its users, and will probably never reach a limit. To discourage users from allocating excessive resources, a metering and billing system is in place. Therefore, it is beneficial to release resources in times of low usage to conserve costs.

In a public cloud, the provider's side, including power costs and hardware wear, is not visible, and the only concern of the user is to minimize the service cost. The billing intervals are an added complexity. They limit the speed at which scaling decisions can be made and still have a positive economic impact. Most of the public services are billed in hourly intervals, some (Google Compute Engine) use shorter intervals.

The cloud is, by definition, tightly integrated with web services, so the changes in capacity may be done through a browser or using an automatic tool that both monitors the application and takes scaling actions through the cloud's API.

In the cloud, processor power and memory are rationed in units of virtual machine instances. There are of course several instance sizes available with every provider, and the hourly cost varies with size. Most providers have fixed sizes, and some allow the customer to specify custom amounts of CPU and RAM (CloudSigma). (But, internally, that complicates the work of the cloud scheduler.)

Other commodities are also billed, such as storage space (the amount of time it is taken is also accounted for), network transfers in and out of the data center or between data centers of the same providers (usually at a cheaper rate), and intensity of disk accesses. These variables may theoretically also be monitored and used in autoscaling, but it is much less common than with CPU cycles. The associated costs are mostly taken as unavoidable (except by a radical change in application architecture).

Because the compute power is billed in total machine-hours, computing a batch job on one instance for 1000 hours will theoretically cost the same as computing it for one hour on 1000 instances. In practice, it is not ideal, as the secondary costs (e.g. for data loading) will be much higher. Some providers (Amazon) also bill an hour for starting a machine. Nevertheless, the possibility is there.

This allows for practically limitless possibilities for application scaling. If we have before insisted that building applications for elasticity be advisable in the private cloud, in the public one, it is an economic necessity. An article by the Czech server Lupa.cz [4] may serve as a cautionary tale. It talks about the economic disadvantages of cloud computing and illustrates it on a study of the migration of a large web service running on four servers with four processors each to the Amazon cloud. The author states that the equivalent computing power would cost ten times the cost of the physical servers in 3 years.

However, the author committed a logical error. His 16-processor server was planned to be the only machine that would be running the service in the next three years. Therefore, it was surely dimensioned to withstand any traffic spike during that period. We infer that most of the time, it will run at a fraction of that maximum capacity. If we dare to say that the average utilization will be 10%, we suddenly see that the tenfold increase in cost

after migrating the application to the cloud would probably be needless, if the application were modified to allow running on a variable number of servers and to employ elasticity. If the author included indirect costs (such as space rental, hardware administration, and upgrades), the outcome of the estimation could turn out to be in favor of the cloud.

It is evident that capacity planning for the Cloud is different from capacity planning for classical deployments. Estimating the maximum capacity is not nearly as important as estimating the curve of the workload in time. Only that way can the cost of a cloud deployment be estimated correctly, without resulting in too large numbers as in the case of the Lupa article, which did not take elasticity into account.

On the other hand, even if a cost calculation showed that the operation of a service is more efficient on owned hardware, it is possible to save on that hardware using cloud services. If we take the performance profile of that service into account and dimension the private cloud or server only for normal load, we get a service that runs on in-house resources most of the time and, in the case of increased demand, it is possible to add compute resources from external sources to cover the spike. This is called cloudbursting and is the performance engineer's way of using the hybrid cloud.

1.4.3 Technical limitations

Here, we made a digression from the reality of cloud technologies towards the idealist image of the Cloud. Unfortunately, the elastic changes of compute resources cannot change CPU power and the amount of memory at runtime. (There are technologies of CPU and memory hotplugging, but they are not being employed in the cloud, and neither is memory ballooning.) It is also not possible to exceed the capacity of one hardware node of the cloud (that would get us into the realm of virtual shared memory supercomputing clusters). Resources are rationed in fixed units of virtual machine instances.

In the cloud, the reaction to high load of some application is not to buy a more powerful server, but rather to run more instances of the server serving the application. When the demand subsides, it is possible to terminate them again to save on costs.

The user can do whatever he or she pleases with these instances, beginning with the installation of their favorite operating system and programs. The programs are usually web servers, database machines, but may also be a scientific simulation or anything else.

As the word "instance" signifies, the virtual machines are mostly created from a template, which allows for quick deployment of larger amounts of servers of the same type. These templates have read-only disks, and the instances are created as disposable clones. The cloud provider usually allows access to a range of predefined images of the most common operating systems, so even OS installation from scratch is very unusual. Customized templates are perfect for quick increases of computing power for an application. It is also possible to save the customization as a script to be run against a clean OS image after start. This scripting, along with access to a central configuration repository, is the core of the DevOps approach to cloud computing.

The time resolution (granularity) of elasticity in IaaS is also limited by the boot time of an instance. This is not only the time to boot an OS, but also includes overhead of

the cloud system, such as copying of the template from central storage, preallocation of temporary disks and set-up of virtual networking. The time range can be from tens of seconds to tens of minutes, depending mostly on the cloud system used, instance size and the cloud controller and storage network load, as measured in a thesis by Klepac [5].

For the user to be able to access all the autoscaled instances, the number of which is unknown in advance, they are added as backends to a load balancer. This is either a stateful NAT (Network Address Translation) device, which assigns each OSI (Open Systems Interconnect) Layer 4 connection to one backend, or, more often, a HTTP (Hypertext Transfer Protocol) proxy, which, as a Layer 7 device, can preserve mapping of application layer sessions to backends using HTTP Cookies. It can also terminate SSL (Secure Socket Layer) connections and buffer server replies, saving server connections (which in some technologies translate to allocated memory) at the backend. The load balancer is usually lightweight, so only one of a small constant number of them are serving an application and act as a single entry point for the users. These are added to the DNS (Domain Name System). Direct addition of the instances there would not work reliably, as they can change often and DNS may be cached at clients.

It is desirable to adapt application development to these IaaS specifics, mainly to the tendency to higher numbers of less powerful servers and to changes in their number, which has higher demands of the developers. It basically amounts to parallel shared-nothing programming. If there is state that needs to be shared between the servers, it should be stored in another tier of the application, perhaps a database server, memory cache or message queue. Luckily, these practices are already common when building web servers for high loads.

1.4.4 Capacity Planning

Classical capacity planning involved the estimation of performance requirements of an application for its whole lifetime. The reason was that the application was tightly bound to the hardware it was to be run on. After the capacity was estimated, a machine corresponding to the requirements was bought, and the application installed on it. It was to remain that way until the scheduled end of life for the machine or the application.

Looking at a daily load graph, such as Figure 1.1, the most interesting point was the peak hour, similar to capacity planning in telecommunications. The peak hour is defined as the interval of one hour, where the number of requests is maximal. The capacity planner would ask for an estimate of the number of users per day from marketing and guess at their distribution throughout the day based on the purpose of the application. Also, any anticipated spikes, such as Christmas sales, were crucial. The server dimensioning was therefore done for the peak hour of the year, plus some allowance for growth during the lifetime of the hardware.

After the demand was known, the application would be benchmarked, and a performance model built so that the demand of users on the application could be translated into the demand of the application on the hardware.

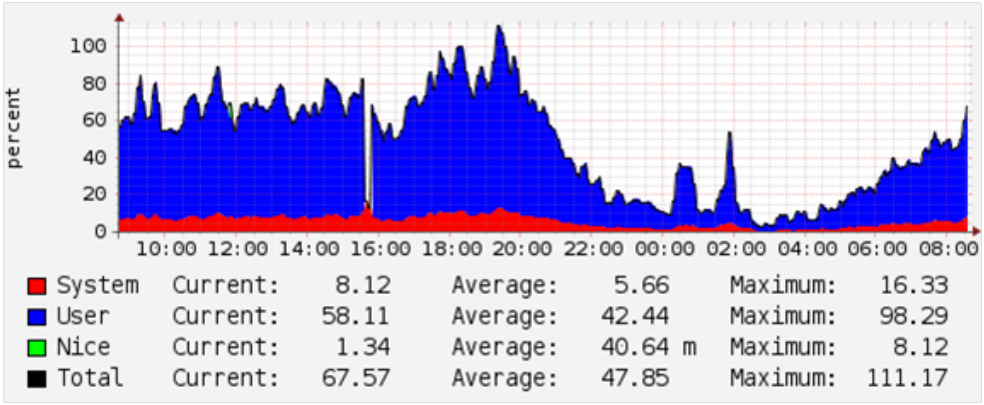


Figure 1.1: Daily CPU load plot of a web server

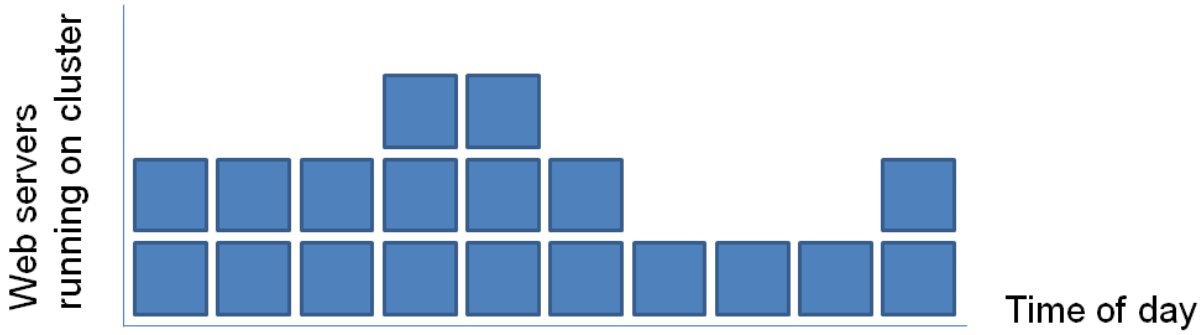


Figure 1.2: Amount of server instances serving a website during a day

In the cloud, this approach leads to unnecessary costs associated with allocated but unused resources. As we can see from the daily load graph, even for a server that is tightly dimensioned for its application, there are times when it is mostly idle. With services, which are serving mostly local clientele, it is during the night.

A more sensible way is to make use of cloud elasticity. For best results with uninformed load balancers (with round-robin or random allocation of requests to backends), the application should be run on a cluster of instances of the same type. The type is chosen from those offered by the cloud provider and should a) be small, as to not waste too many resources during the times of low demand and b) have a good performance/cost ratio for the application. With scaling in effect, the load graph is basically quantized by the number of instances serving the application, like in Figure 1.2. (This is then the load as seen by the cloud provider, who counts the allocated resources and does not see inside them.)

The marketing step is not really necessary from the dimensioning standpoint because, with scaling, we do not need to know the peak demand in advance. However, due to the pay-per-use model, it is much more necessary for cost estimation. The benchmarking step is more important, with the goal to find out the right type of instance for the application.

The providers usually have a main line of instances, where the CPU/memory ratio is fixed and increases by a factor of two (which leads to more optimal scheduling of instances to physical machines on the provider's side), but some also have lines with more CPU and less RAM or more RAM and less CPU for the same price.

1.4.5 Automatic scaling

It is impractical to set the amount of virtual machines serving an application by hand, because the workload profile might change. If, at any time, the resources are underprovisioned, it will result in a decrease of GoS (Grade of Service) for the end user (the GoS here is expressed as the response time distribution of the web application), or in the worst case, in its unavailability. Overprovisioning leads to wasted costs. An autoscaler is able to manage the amount of instances automatically.

Autoscaling is in this case defined as the ability to increase or decrease the number of application instances serving an (in most cases) interactive workload, whose characteristics are changing in time. It is dependent on performance metric monitoring and a cloud API (Application Programming Interface) to effect the changes in the number of instances and load balancer configuration. An equivalent term is self-adaptive load-balancing.

In the Public Cloud, autoscalers are either available from the provider or as a third party service. They offer the possibility to run a small number of application instances persistently and to boost the computing power when the offered load demands it. Some private clouds (OpenStack, Eucalyptus) have already integrated autoscalers as well. It is also possible to use an autoscaling system that can be deployed in a virtual machine in the Private IaaS Cloud and is able to automatically manage instances of other applications on it. One such application is ScaleGuru.

Private Clouds are good for experiments as those do not cost money and the reproducibility is much better than in the shared environment of a public cloud. The problem is that they do not offer the same level of services yet. Particularly, automatic scaling is necessary if data about variable load are to be collected for forecasting experiments. At the time of writing of the ScaleGuru tool, the autoscalers for Eucalyptus and OpenStack were not yet ready. This autoscaler still has the added benefit of simplicity and expandability. It also has integrated monitoring and performance logging.

Due to the metered approach to billing in Cloud Computing, some monitoring is always available. For the purpose of billing, a granularity of one hour is perhaps enough, but for automatic scaling, a finer granularity is needed. Practical autoscalers operate on 1, 5, or 15 minute average values. The finer the monitoring granularity, the faster the reaction to a high load situation, but also the risk of oscillations.

The monitored metric varies by implementation. An autoscaler may use latency (as the primary GoS indicator) directly (as in Google App Engine), but in IaaS it most often operates on a lower level of CPU load, disk and network measurements (such as in Amazon Web Services). There are also Autoscalers working on load balancer queue length (OpenShift). The client sets the scaling thresholds for these parameters based on his or her understanding of the application and its performance model.

The thresholds work similar as performance alarms in traditional monitoring tools. For example, an alarm could be defined on the the CPU load metric and would fire if the average value from the instances serving an application was over 70% for 5 minutes. The result of this alarm would then be defined by a policy, such as to add a fixed number of instances or an amount expressed by a percentage of currently running instances. Using multiple rules, it is possible to create a dynamic response curve. There is also a setting for the minimum and maximum number of instances, as a safety measure against oscillations (the lower limit) or performance runaway errors (the upper limit).

1.4.6 Predictive scaling

Scaling the amount of resources dedicated to an application depending on the workload can lower the costs of the infrastructure, as the capacity does not have to be provisioned for peak loads, which was the case in traditional static virtualization. However, that means that quick reaction to an increase in demand for an application is highly desirable so that the users will never feel that the provider is saving on server capacity.

The available scaling services are mostly reactive – they react to the measured level of resource utilization in the virtual machines or to the measured latency of the application.

The downside of reactive autoscaling is that the provisioned capacity lags after the demand, meaning that extra resources will get allocated after an overload situation has started. There is a trade-off between overprovisioning and the possibility of SLO (Service Level Objective) violation, which is controlled by setting the upscaling threshold of the autoscaler. In downscaling, the reactive approach is prone to oscillation, which happens when the system load goes back over the upscaling threshold a short time after resources have been released.

If proactive scaling methods employing load forecasting were used, the autoscaling service could avoid overload situations, and thus the GoS (Grade of Service) could be raised and prevent oscillations that can arise in reactive autoscaling.

Workload forecasting can be used to predict a daily curve of the autoscaled service. A pro-active autoscaler with forecasting can then mitigate both the provisioning lag and the oscillation. It still has to be combined with reactive scaling in case of an unpredictable spike. An example of this approach can be found in an article by Moore, Bean, and Ellahi [6].

The demand can take an unpredictable leap for example as a reaction to a marketing campaign, and if the capacity of the application is not sufficient to cover the spike in request intensity, it means an increase in response time or temporary unavailability of the application. That can not only endanger the campaign but the reputation of the company because some impatient customers may never return and spread the tale of the broken website to others.

This problem is addressed by anomaly detectors that can detect these spikes early and also use prediction techniques. In reality, the spikes caused by human activity are rarely abrupt, but have a distinct onset, which can be detected.

A survey of prediction techniques for autoscaling is available from Weingartner, Brascher, and Westphall [7] and Lorido-Botran, Miguel-Alonso, and Lozano [8].

1.4.7 Evaluation and Metrics

In order to evaluate different autoscaling algorithms or threshold settings, we need to have measurable metrics to compare them. Firstly, when comparing autoscalers, we need to run them on the same workload. That means either to set up an experiment with several load-balancers and backend pools, to which the same traffic is mirrored, or to use simulation. It is next to impossible to find someone with real traffic who would allow scientists to run autoscaling experiments. The service providers are mostly concerned with risks of unavailability of their applications, and then there are concerns with request data confidentiality. In the end, simulation is practically the only choice.

Even request intensity traces, which are in fact just time series graphs with no confidential data in them, are very hard to come by. There are a few traces in the Internet Traffic Archive[1], but they are more than 15 years old, mostly too short to perform learning of forecasting algorithms, and of anomalous traffic (like a football match or Olympic games), not well predictable steady state traffic.

Studies of autoscaling techniques have been criticized for a lack of verifiability because they are not evaluated on publicly available workload traces (i.e. in a survey by Weingartner [7]). The problem in the field of Cloud Computing is that while the Grid is mostly operated by academic institutions, which are not afraid to share their data (see the Grid Workload Archive (GWA) [9]), the Cloud is run mostly by commercial subjects, which are bound by various privacy agreements and will not publish their data. We have tried to obtain it and failed, as have probably most of our colleagues cited in the survey. Scientific clouds, which are indeed in existence, do not help the matter, as their workload is mostly comprised of high-performance computing tasks not suitable for the Grid, and not web traffic. This was observed e.g. on MetaCloud[2].

Moreover, the available cloud traces (we know only of the Bitbrains 2012 trace from GWA) are of server utilization, and autoscaling simulation using QN models requires a request intensity trace, simply because by introducing autoscaling, one is changing the utilization and trying to keep it inside some preset bounds. The data from the trace is made obsolete after the first scaling action. Much more interesting data would be that collected from logs of individual services, not basic system monitoring data available to a cloud provider, who does not see inside clients' virtual machines.

Along with the request intensity trace, a simple performance model is necessary to be able to translate the requests to load on the backend servers. With that, it is possible to create a baseline. That would be the cost of running the application on a constant number of instances which are enough to serve the peak hour.

[1] `http://ita.ee.lbl.gov/`

[2] `http://www.metacentrum.cz/en/cloud`

The running cost can be expressed in money, but it is sufficient to use time units to measure e.g. the number of instances times the number of hours each of them was running. The cost of the baseline will be the greatest and different autoscalers/threshold settings can be ordered by the cost reduction they achieve.

The autoscaler works by starting and stopping virtual machines. Doing it too frequently is undesirable, because, with some public clouds (Amazon), starts and stops are billed. In the private cloud, it creates extra load on the cloud controller nodes, storage subsystems, and network. Also, there is a delay after starting a virtual machine before it is ready to serve requests. Therefore, start/stops caused by the autoscaler should be measured and used as a measure of its performance. Too many signify some kind of instability of the algorithm.

Because the goal of autoscaling is to minimize cost while keeping the end user GoS within bounds, a way of quantifying the GoS is needed. We have defined it as the latency distribution of the application, but distributions are not easy to compare. It is necessary to either take a quantile from them, which will tell us the maximum time that, e.g., 95% of the users will see, or to set a GoS goal in time units and ask for the percentage of users that will see that time or less.

Both versions are used in practice. The first choice is good for graphing a live system, but if we keep to the definition of autoscaling optimization from the previous paragraph, we see that the second variant is more suitable for specifying boundary conditions (or SLAs in business).

1.4.8 Queueing Theory

The queuing theory was founded at the beginning of 20th century by A.K.Erlang to calculate the capacity of Danish telephone exchanges. It has application not only in telephony, but also in transportation, manufacturing, and customer service[1]. It has also been applied to computing, mainly for server dimensioning in client-server architectures. On the Internet, most traffic (excluding file sharing peer to peer services) is due to the HTTP protocol. The World Wide Web carried on it is also based on the client-server model and with dynamic web pages, most of the intelligence and computational complexity is on the side of the server. That places high importance on server dimensioning so that adequate response time is guaranteed for all users, who may be using the server at once. Queuing theory is one of the best instruments for that.

When deploying a web application, it is necessary to correctly estimate its demands for compute resources, such as processor time, operating memory size, storage space and network throughput, and to choose the best way to provide the infrastructure, with respect to cost, reliability, and room for growth.

Queueing theory works with exactly these terms. It defines the relation between request arrival rate, response time, throughput and utilization, enables the creation of models of availability, reliability, and security (if we want to predict breach frequency and time to

[1] `http://en.wikipedia.org/wiki/Queueing_theory`

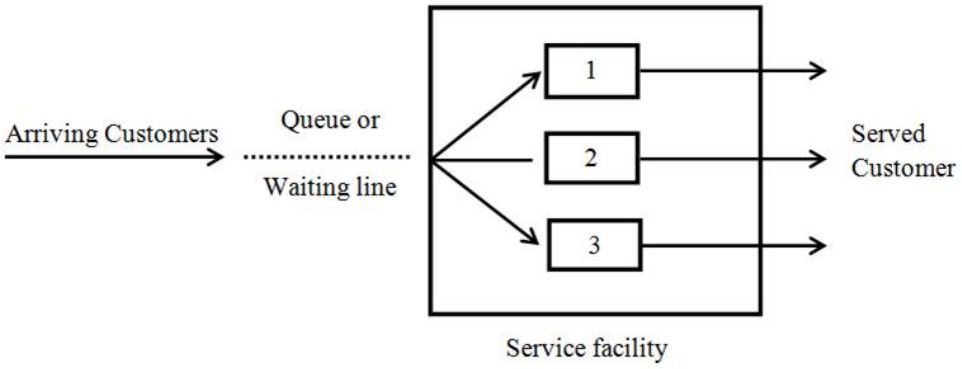


Figure 1.3: Diagram of a queueing system

resolution). Every performance engineer needs to know basic conclusions on lower bounds on service time depending on service demand and how to calculate maximum throughput from queue length and response time.

Queueing theory is based on the model of a service system, which consists of an input flow of service requests (from a finite or infinite amount of users) and a number of service lines that serve these requests. Both the input flow and the service times of the lines are stochastic processes, and the theory can analytically provide answers to questions like what is the distribution of waiting times of the requests in a queue, or what is the probability of denial if queueing is not allowed. Inversely, knowing SLOs, one can compute the number and speed of service lines necessary to achieve it with a certain probability.

Other components of the service system (see Figure 1.3) are the already mentioned queue, a control system, which assigns requests from the queue to the service lines. The serviced customers constitute the output flow, and there may also be a flow of rejected requests. Alternative names are the offered flow, the transferred flow, and the error flow. A request may be rejected when all the service lines in a service system without a queue are busy, when these queues are full if their maximum capacity was defined, or when a maximum waiting time is exceeded.

The input flow needs to satisfy the following three conditions[1]:

- Stationarity – the character of the input flow is constant in time and the system being described will eventually reach a statistically steady state.
- Ordinarity – in a given instant, there is no more than one incoming request. The flow can be described using interarrival intervals. (Although batch arrival models also exist, they will not be discussed here.)
- Independence – request arrivals and service times are statistically independent.

[1] `http://ecoursesonline.iasri.res.in/pluginfile.php/5591/mod_resource/content/1/Lesson_16.htm`

The most important parameter, for which the queueing models are solved, is request satisfaction, which can be expressed in percent, or quality thereof (e.g. waiting time in a queue). Quantitative parameters are request intensity expressed as the number of request per unit of time, and operational load intensity, which is expressed in erlangs. One erlang is the request intensity that can be transferred by one service line working at 100% utilization.

As queueing theory has its roots in statistics, various types of service systems are mostly differentiated by statistical distributions of interarrival and service times, and also the number of service lines. D.G.Kendall designed this classification and also its shortened notation:

According to the introduction at [10], the type of a service system is given by a three-tuple A/B/n, where A is the distribution of interarrival times, B the distribution of service times and n the number of service lines. An extended version of the notation can also be used, which adds a fourth element – the maximum size of the queue, and a fifth one – the queue discipline. A and B may be replaced by one of:

- D – deterministic (constant time)
- M – Markovian (Poisson arrival process or exponential interarrival times)
- G – General (specified by mean and variance)
- GI – General Independent
- Ek – Erlang with parameter k, and perhaps others

The queue discipline may be:

- FIFO (First In First Out, or also FCFS (First Come First Served))
- LIFO (Last In First Out)
- PS (Processor Sharing, i.e. all concurrent requests are being processed at once with proportionally extended service times)
- Random

To describe the arrival of requests into the service system, the Poisson process is most often used. It describes real-life situations with adequate precision and is easy to solve analytically (using it removes the need for step-by-step simulation). It describes requests coming from a memoryless stochastic process. The requests are generated randomly with exponential interarrival times. Memorylessness is a property of the exponential distribution and means that the probability, with which the source generates another request, is independent of the time or quantity of requests in the past. The request intensity can be regulated by setting the mean of the exponential distribution.

If the requirements on the input flow are satisfied, the observer at the input of the service system will see a flow, whose number of requests per unit of time follows the

Poisson distribution. The interarrival times at a queue, which is fed by several exponential request generators, are governed by the Erlang distribution. If the service times are also exponentially distributed, the output flow has the same properties as the input one.

The load or utilization is one of the most important and least understood quantities of queueing theory (at least by people specializing in computing). It is dimensionless, because it is defined as time by time, or more comprehensibly, the busy time divided by the total measurement period, i.e.

$$U = \frac{B}{T}$$

That also means that there is no utilization at a distinct point in time. The service line is either busy or not busy. In the case of a single service line, the utilization is equal to the probability that a request will have to queue because the line was busy.

One of the most interesting results are the lower bounds on latency and corresponding higher bounds on throughput. These are asymptotes that a system defined by its service demand, i.e. the minimum service time, may never exceed. There exists a lower bound for a lightly loaded system, which is equal to the service demand, because the service line was mostly free, and the requests were served immediately, and one for a heavily loaded system, which has utilization near 1, and whose service time is linearly increasing with the arrival intensity and reflects the time the requests are waiting in queue. See e.g. Menasce[11] for the formulas. In reality, there is a knee between these two modes of operation, see Figure1.4[1].

1.4.9 Queueing Networks

These basic service system models may be chained together into a graph to create more complex queuing network models. They can, for example, model a situation where a request will be first served at the CPU, then with a certain probability will wait for a hard disk, then return to the CPU queue, and finally go to the network card to be sent out, such as in Figure1.5[2].

When all the input flows and service times are Markovian, it is possible to analytically solve the model by transforming it to a Markov chain. The downside is a quick expansion of state space of the chain. There are also other solution methods, such as the MVA (Mean Value Analysis), which have lower complexity. MVA works inductively, by modeling a network with zero requests, and a network with $M + 1$ requests, until the desired load is achieved. For all the relevant algorithms, see [12]. Concretely, MVA for a closed queueing

[1] http://perfdynamics.blogspot.nl/2010/03/bandwidth-vs-latency-world-is-curved.html
[2] http://www.simalytic.com/CMG96/CMG96htm.htm

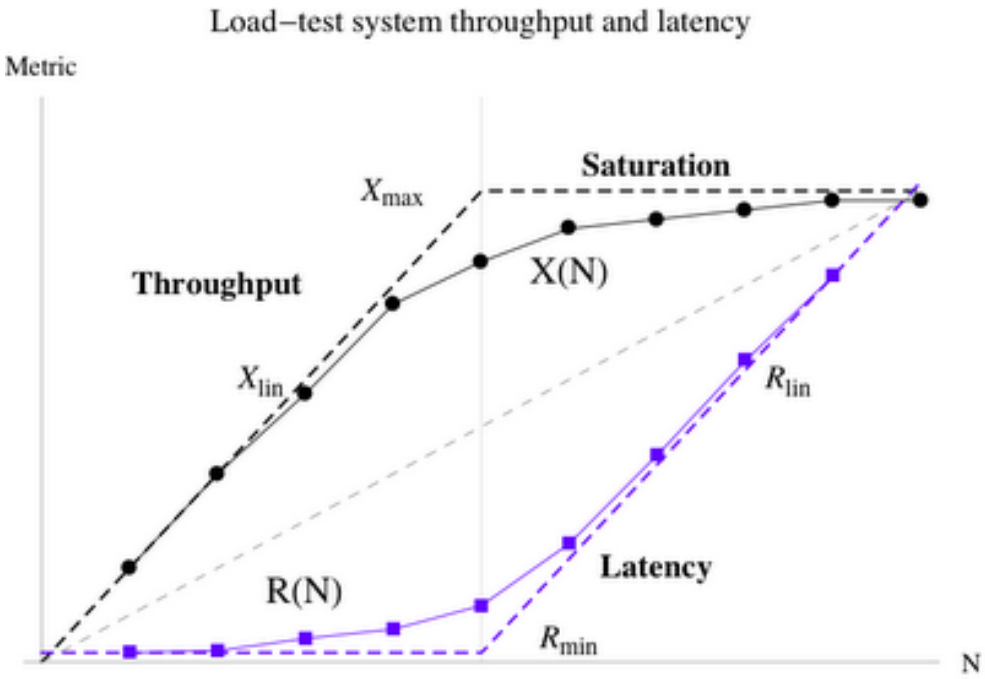


Figure 1.4: Throughput-latency eye diagram

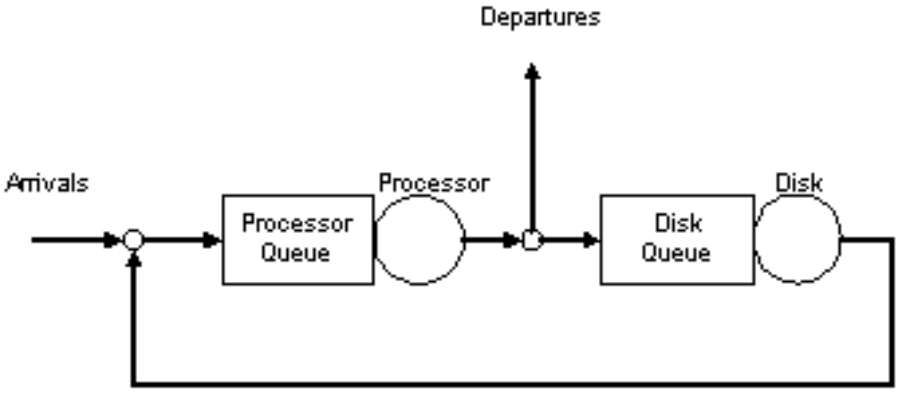


Figure 1.5: A simple queueing network describing a database system

network looks like this:

$$W_k(m) = \frac{L_k\,(m-1)+1}{\mu_k} \tag{1.1}$$

$$\lambda_m = \frac{m}{\sum_{k=1}^{K} W_k(m) v_k} \tag{1.2}$$

$$L_k(m) = v_k \lambda_m W_k(m) \tag{1.3}$$

where $m = 1..M$ is the number of customers in the system, $k = 1..K$ denotes the nodes of the network, W are waiting times, L are queue lengths, λ are arrival rates, μ are service rates, and v are visit counts of customers to each node.

There are two types of queueing networks, as well as of single service systems, namely open and closed. In an open model, requests enter the system, traverse the network, and exit. In a closed system, there is a constant number of requests in the system, which are most commonly buffered at a dedicated service station, where the clients "think". This model is most commonly used in interactive systems where the clients alternate between thinking and waiting for service. That means that the arrival rate from the buffer station into the rest of the system gets lower when the clients are queued elsewhere. It is a self-regulating system.

Please note that the diagram of the lower bounds is valid for a closed system. In an open system, when the utilization reaches 1, the queue length and service time approach infinity because the model describes a statistical equilibrium and cannot model the slow rise of the queue length that would happen in practice. The input flow of an open system is characterized only by the arrival rate, whereas that of the closed system by the number of clients and think time. A batch job system works similarly to an interactive system; only its think time is zero because requests are issued from a batch job queue (which is deemed infinite).

In particular for modeling autoscaled web servers in the cloud, we find that queueing networks are a great abstraction because they can model multiple service stations, their number equal to the number of servers in an application tier. Secondly, chaining the service stations together can model multiple tiers. Using routing probabilities, one can model that e.g. between a web server and a database server, not every request from the user will make it to the database, or, the other way around, that one web request will result in several requests to the database (as is the case with our favorite benchmark application, Wordpress).

Also interesting is the ability of most simulation tools to work with multiple client classes with distinct flow rates and service demands. This is useful when the real situation that is being modeled has a multimodal service time distribution, as is the case with web servers which serve both dynamic and static content.

The downside of queueing network models is that they model the system in a stochastic steady state. It is impossible to capture individual requests in the model, only time interval averages. However, this plays well with practical performance monitoring tools, which sample the measured system and save averaged values in a time series database. It is

also not possible to solve an overloaded open queueing system using the modeling tools. However, extrapolation using simple formulas is found to sufficiently capture the slow rise in queue length and service time and the model is able to carry on when the overload subsides.

1.5 Structure of the Dissertation Thesis

The thesis is organized into five chapters as follows:

1. *Introduction*: Describes the motivation behind our efforts together with our goals. There is also a list of contributions of this dissertation thesis and theoretical background.

2. *State-of-the-Art and Related Work*: Introduces the reader to the necessary theoretical background and surveys the current state-of-the-art in two areas, those being *Performance Prediction* and *Autoscaling Simulation*.

3. *Overview of Our Approach*: Summarizes the main points and methods employed in this research.

4. *Main Results*: Is further divided into three chapters that detail the experiments performed and their results, namely *Fesibility of Interactive Workload Prediction*, *CloudSim Modification for Interactive Traffic*, and *Cloud Simulator based on Queueing Theory*. All three have been previously published in reviewed journals.

5. *Conclusions*: Summarizes the results of our research, suggests possible topics for further research, and concludes the thesis.

Chapter 2

State-of-the-Art and Related Work

2.1 Performance Prediction

This section presents a study of scientific articles concerning performance prediction and cloud computing. Although load prediction articles existed for a longer time, mainly dealing with grids or server clusters, the keyword cloud computing limits the results to no older than 2011. About 250 article abstracts have been examined, of those 20 were found highly relevant and were studied in detail. The most articles (10) were concerned with automatic scaling, 9 dealt with performance simulation, 6 with resource allocation, 4 with task scheduling, 4 with anomaly detection and 3 presented new forecasting methods. As to the methods identified, time series analysis was performed using statistical means (averaging and regressive methods), using machine learning (neural networks and Markov models), or by pattern matching. Other prediction approaches that are not time series analysis methods include supervised and unsupervised machine learning, queueing theory models, control theory controllers, game theory optimization and general heuristics. This literature research should give a good overview of prediction tasks that are present in clouds and methods available to solve them.

This section has two main subsections, Subsection 2.1.1 presents the research from the view of the research goals (e.g. saving power, optimizing resource allocation), Subsection 2.1.2 views the works from the standpoint of used methods (time series analysis, machine learning, etc.), and the last subsection is Summary.

2.1.1 Research topics

This subsection attempts to classify the studied articles by the main topic of the work. Some of them may use prediction only marginally or use another form of prediction than performance forecasting.

2.1.1.1 Forecasting methods

Several articles present new forecasting methods and compare them with others using workload traces. There are several such traces that are widely cited, most notably World Cup 1998 web request trace and Google cluster batch workload trace. These are available in [13] and [14], respectively.

Articles in this category are [15], which presents a method to enhance the Network Weather Service [16], a prediction framework from 1999, which currently seems to be discontinued. Article [17] presents a method from the area of pattern matching, see next section.

An interesting work is by Herbst [18], which holds a summary of forecasting methods implemented in the R statistic language, along with their strong and weak points, and presents an algorithm for automatic selection of method based on prediction goals and, to a lesser degree, input data. It contains, by increasing complexity, moving average, simple exponential smoothing, Croston's method, cubic smoothing, ETS, ARIMA, SARIMA and tBATS.

2.1.1.2 Resource allocation optimization

A popular theme in cloud computing is placement optimization in all its varieties and goals. Most prevalent is the drive towards Green computing, that is consolidating workload to as few machines as possible while maintaining SLA (Service Level Agreements). Another optimization problem lies in hybrid clouds. There are works that try to compute best routing of requests between multiple distributed replicas of a server, perhaps in a multi-tier or mash-up infrastructure. Also cost vs. latency optimization between providers is an open problem.

Articles from this class mostly don't contain any forecasting methods, rather optimization algorithms such as support vector regression and genetic algorithm used to compute VM to PM allocations in [19].

2.1.1.3 Task scheduling

If this research wasn't focused on cloud, but rather on grid, this would be the prevalent class of articles. Even in the area of cloud, there were 4 relevant articles that dealt with optimization of batch job queues. The quantities predicted in this field are several. Firstly, to optimize a batch job system, the run time of a task needs to be known as precisely as possible. With job durations estimated, one can predict the run time of the entire queue and if the jobs also have deadlines, one may compute the necessary amount of resources for the jobs to complete on time.

In the context of cloud, this can be used to automatically start a cluster of the right proportions. This approach is, arguably, outdated, since in cloud computing, the amount of resources is ideally changed dynamically, not allocated once at the beginning. Some classic cluster computing tools, sadly including the currently popular Hadoop, are difficult to scale, so this static sizing approach is still used.

Zhang in [20] uses model predictive control and ARIMA to predict task durations and then optimizes the cluster cost between the electricity costs and costs for deadline violations. An interesting observation is that he first fitted the model to the data, and after that specified the forecast horizon as the time when the accuracy of the method fell below that of the naïve predictor (which copies the last observed value). The reported horizon was 33h in 5 minute steps. Eldin [21] presents a simulation of a reactive-predictive autoscaler for compute clusters using control theory.

2.1.1.4 Automatic scaling

Automatic scaling is currently one of the most prominent topics in cloud research, both academic and commercial. Scaling the amount of resources dedicated to an application depending on the workload can lower the costs of the infrastructure, as the capacity doesn't have to be provisioned for peak loads, which was the case in classical static virtualization. The downside of reactive autoscaling is that the provisioned capacity lags after the demand, meaning that extra resources will get allocated after an overload situation has started. There is a trade-off between overprovisioning and the possibility of SLO (service level objective) violation. In downscaling, the reactive approach is prone to oscillation, which happens when the system load goes over the upscaling threshold a short time after resources have been released. Automatic scaling employing some kind of load prediction could mitigate both these problems. It still has to be combined with reactive scaling in case of an unpredictable spike. This problem is further addressed by anomaly detectors that can detect these spikes early and also use prediction techniques.

Overloaded infrastructure Specific subproblems within this class are how to deal with overloaded infrastructure, where the peak workload is at or over the cluster capacity. It is then possible to dynamically migrate VMs to less loaded PMs or to actuate allocation of CPU cycles or VM priorities (depending on hypervisor), so that the more important services don't suffer. This is also an optimization problem. When the load of a VM hits the cap, the prediction is likely to be inaccurate, as it captures the amount of resources used, not the resource demand.

Hybrid cloud with spillover Two articles also dealt with a hybrid cloud, where automatic scaling works in longer intervals and traffic may spill over from one location to the other in the case of a traffic spike. One of them is [22], which uses simple exponential smoothing as its predictor. The downside is that the spillover traffic coming from the other site is rather unpredictable (this is also a hard problem in queueing theory).

Green computing There is also a tendency towards green computing in this category. There are two or three approaches to lowering the power draw of a virtualization cluster. The first is powering machines on and off, which has the downside of a longer reaction time and higher possibility of node failure. The time to turn the machine back on can

be significantly lowered by using suspend to RAM. The second approach is to use DVFS (Dynamic Voltage and Frequency Scaling) to lower the energy required by the CPU. The downsides here are that the savings are much lower, as most of the power used by the servers is static and not dependent on CPU load. Secondly, response times of application running on downscaled processors can be slightly higher.

The first category is represented by three articles. In [23], a queueing network simulation is created that monitors throughputs and queue lengths at load balancers and reacts by adding or removing VMs every 10 minutes and powering on or off machines every 2 hours. It doesn't contain an actual forecasting method. Article [24] develops a pattern-based prediction method that decides the number of needed machines in the next 5 minutes. The authors of [25] use a neural network to exactly the same ends.

A proponent of DVFS is Shen with the CloudScale system [26], which uses both migration and frequency scaling to lower the power usage of a cluster. It uses a Markov chain predictor on FFT signatures and predicts 10s ahead for frequency scaling and 100s ahead for migration. It is designed for the Xen hypervisor and uses its credit CPU scheduler to dynamically cap processor and memory usage of VMs in order to ensure good multi-tenancy. The article also describes how to deal with underestimation errors in prediction and with prediction after overload situations.

Both DVFS and power actuation is used in [27], where Holt's double exponential smoothing is used with a frequency of 5s and forecast horizon of 10 minutes. The cluster used in the experiment was heterogeneous, so the authors pre-computed a table of which servers and at what frequency should be running for different workload intensities. In the evaluation, a server was started of stopped tens of times per hour.

In public cloud The previous articles in this category dealt with a private cloud, virtualization or bare metal cluster. They used this fact to control both the client a provider side of the IaaS model from one application. In public cloud, the provider side, including power costs and hardware wear, is not visible and the only concern of the user is to minimize the service cost. An added complexity are the billing intervals, which limit the speed at which scaling decisions can be made and still have a positive economic impact.

Only one predictive autoscaler for public clouds was found in the articles, as part of a complete automated load and scalability testing framework by Vasar [28]. The autoscaling component uses Holt's double exponential smoothing to predict the workload, which is however rather specific as it is driven by a ramp-up traffic pattern. The scaling granularity of the system is 1 hour as dictated by Amazon EC2 billing. All instances are started after full hour and terminated minutes before, to work around the problem of selection of the instance nearest to full hour. The predicted quantity is requests at the load balancer, which is converted to load by a simple queueing theory model.

2.1.1.5 Performance simulation

Another class of articles found during the study deals with queue theoretic models of applications in the cloud. Mostly, the applications are first profiled, a performance model

is constructed, and an estimate of how many nodes are required for a certain load on the cloud infrastructure is given. Four articles dealt with server applications, four with sizing of a compute cluster, including Hadoop, one article presented a profiling and simulation tool.

The article by Iverson [29] deals with heterogeneous grid computing and statistical prediction of task execution times in this environment. A benchmark of different machine types and workloads has been conducted and analyzed by k-means regression. The model could then estimate the run time of a profiled task on different machines. More articles of this kind have not been studied as performance models were not the goal of this analysis.

2.1.1.6 Economic models

Prediction can also be used from the perspective of a public cloud provider. For example, the price of a certain commodity may be set based on demand. If the demand is predicted in advance, the so called spot price can be set more accurately than by manual means, and thus regulate the availability of said resource by market mechanics. One article of this kind was found, but not investigated beyond the abstract.

2.1.1.7 Anomaly detection

Anomaly detection is an approach that utilizes a prediction method over a small forecasting horizon and raises an alert when the actual value is significantly different from the prediction. It is already employed in commercially available system monitoring tools. It augments usual availability measures and performance threshold alerts and is able to point out to the administrator that something is not behaving normally. In contrast to threshold alerts, it can also alert when the usage of some resource is exceptionally low, which may for example mean that users are unable to access the site or have been redirected elsewhere by an attacker.

A good example of anomaly detection is in Dean [30], who uses self organizing maps to detect when the system parameters get outside their normal parameters. The author claims this is a prediction technique, in contrast to detection, but the effect is actually closer to detection of pre-failure states and works in the horizon of tens of seconds. The PREPARE system [31] uses a 2-dependent Markov model to track the monitored system in discrete performance steps and uses a naïve Bayes tree to classify the states as normal or abnormal. The system has means to prevent a predicted anomaly by migrating virtual machines or increasing their memory or CPU allocations.

The literature search also contained two articles on anomaly source identification in multi-tier systems, but they contained no predictive algorithms per se.

2.1.1.8 Low-level Infrastructure performance

As the modern computers are becoming so complex that the execution of a program is an unknown quantity and must be evaluated by statistical means, there is a field of computer science that studies factors that influence the run time of a program. In virtualization,

this is most notably scheduling of VMs to CPUs and sharing of the memory hierarchy by workload that runs inside the VMs. This study discovered one article that deals with profiling VMs and predicting slowdowns stemming from co-location of different VMs.

2.1.1.9 Cloud service optimization

Beside users VMs, a cloud may also contain shared services, such as databases. If these are to serve multiple customers, it must be ensured that one user cannot saturate the service and make it unavailable or poorly performing for others. One article shows an approach that places a proxy in front of a shared database and, based on a prediction of duration of incoming requests, only admits requests that won't overload the service.

2.1.2 Employed methods

In this subsection, the prediction or classification methods employed in the studied articles are classified and discussed in more detail.

2.1.2.1 Statistical time series analysis

A time series is a set of observations of a certain quantity that are ordered in time. For the purpose of computer system monitoring, the time intervals are mostly equidistant. The purpose of time series analysis is most of the time to create a forecast of the series in the future. All forecast methods may be compared to the so called naïve forecast, which is equal to the last observed value. A time series can be analyzed by several methods, the most thoroughly studied are from the area of statistics.

One of the useful tools it provides is time series decomposition. A time series is formed from three main components, the first is a seasonal pattern, which may or may not be present. It is most prominent if the data contains a daily, weekly, etc. cycle of natural or human behavior. In computer systems, seasonality is mostly created by interactive user activity and the shape of the cycle depends on the nature of the system, e.g. whether it is a business system or one that people mostly use in their free time. The second component is trend, which signifies the slope of the time plot in a longer term. It may be modeled by linear segments, splines, or simply as a moving average of the data with the seasonal effect already removed. The remainder of the data after the first two components are subtracted is the error term. Ideally, when the time series is a stationary stochastic process (a requirement of most statistical analysis methods), the error term is white noise. The seasonality and trend may be of additive or multiplicative nature.

Averaging Time series models that may be used for forecasting are built either on moving averages or on autoregression. The most basic averaging model is the simple moving average. It takes a sliding window and runs it over the time series. It produces a smoothed time series by substituting the average of the observations in the window for

the measured value. Its forecast is the average of the last window. Used as the cheapest algorithm of the adaptive approach in [18].

Simple exponential smoothing gives more weight to more recent observation while having infinite memory. It is implemented as one variable that holds the average, and is updated by each new value with a given weight α, while the last value of the variable has weight $1 - \alpha$. The smoothed value s at time t is given by the formulas

$$s_0 = x_0 \tag{2.1}$$
$$s_t = \alpha x_t + (1 - \alpha)s_{t-1},\ t > 0 \tag{2.2}$$

where x is the original value of the time series.

It can also be used to smooth a time series, the forecast is the last value of the average, so it looks like a straight line in the mean. It is good for time series that do not exhibit trend or seasonality. The weight can be calculated automatically from training data using linear least squares. Used in [18] and in [22] for automatic scaling.

Double exponential smoothing or Holt's method improves exponential smoothing by adding a second variable holding the difference of the current value from the smoothed average, which captures the slope. The slope is calculated as an exponentially smoothed difference between the current value and predicted mean. It also has 2 weight parameters, α and β. This way, the trend of a time series is captured. It is given by the formulas

$$s_1 = x_1 \tag{2.3}$$
$$b_1 = x_1 - x_0 \tag{2.4}$$
$$t \geq 2 : \tag{2.5}$$
$$s_t = \alpha x_t + (1 - \alpha)(s_{t-1} + b_{t-1}) \tag{2.6}$$
$$b_t = \beta(s_t - s_{t-1}) + (1 - \beta)b_{t-1} \tag{2.7}$$

where b_t is the best forecast of the trend. The forecast at time t plus m steps into the future is given by

$$F_{t+m} = s_t + mb_t \tag{2.8}$$

It can be used for short term forecasts or when the time series is not seasonal. The forecast looks like a line at the average with a slope equaling the trend. Used in [28] for predicting the effect of a ramp-up traffic pattern and in [27] for turning on/off servers in rather short intervals.

Croston's method was created for forecasting time series of intermittent demand. It splits non-zero and zero parts of the series and performs forecasts separately using simple exponential smoothing. Referenced in [18].

Cubic smoothing uses a linear forecast function to extrapolate a steep trend. Good for short term forecast on non-seasonal data. Referenced in [18].

Triple exponential smoothing or Holt-Winters method extends Holt's method to capture seasonality. To this end, an array of seasonal coefficients is added. The seasonal memory

array holds the factor or addend (depending on whether multiplicative or additive seasonality is used) of each observation point in the season to the exponentially smoothed value, and is itself updated through exponential smoothing. All points use the same smoothing weight, γ. The formulas look like this for multiplicative seasonality:

$$s_0 = x_0 \tag{2.9}$$
$$s_t = \alpha\frac{x_t}{c_{t-L}} + (1-\alpha)(s_{t-1} + b_{t-1}) \tag{2.10}$$
$$b_t = \beta(s_t - s_{t-1}) + (1-\beta)b_{t-1} \tag{2.11}$$
$$c_t = \gamma\frac{x_t}{s_t} + (1-\gamma)c_{t-L} \tag{2.12}$$
$$F_{t+m} = (s_t + mb_t)c_{t-L+1+(m-1) \mod L} \tag{2.13}$$

and for additive seasonality:

$$s_0 = x_0 \tag{2.14}$$
$$s_t = \alpha(x_t - c_{t-L}) + (1-\alpha)(s_{t-1} + b_{t-1}) \tag{2.15}$$
$$b_t = \beta(s_t - s_{t-1}) + (1-\beta)b_{t-1} \tag{2.16}$$
$$c_t = \gamma(x_t - s_{t-1} - b_{t-1}) + (1-\gamma)c_{t-L} \tag{2.17}$$
$$F_{t+m} = s_t + mb_t + c_{t-L+1+(m-1) \mod L} \tag{2.18}$$

where c is the array of seasonal correction factors and L is the length of the cycle. The weights are often initialized like this:

$$b_0 = \frac{1}{L}\left(\frac{x_{L+1} - x_1}{L} + \frac{x_{L+2} - x_2}{L} + \ldots + \frac{x_{L+L} - x_L}{L}\right) \tag{2.19}$$
$$c_i = \frac{1}{N}\sum_{j=1}^{N}\frac{x_{L(j-1)+i}}{A_j} \quad \forall i = 1, 2, \ldots, L \tag{2.20}$$
$$A_j = \frac{\sum_{i=1}^{L} x_{L(j-1)+i}}{L} \quad \forall j = 1, 2, \ldots, N \tag{2.21}$$

where N is the number of complete cycles in the training data.

The forecast contains the seasonal pattern repeated over and over, superimposed on the trend, starting at the average. It can be used for medium range forecasts and for seasonal series. There is also a function called dshw() (for Doubly-Seasonal Holt-Winters) in R, which can capture two combined seasonal effects, such as daily and weekly. It is not referenced by any examined article, however [18] cites ETS (Extended exponenTial Smoothing), which is a generalization covering simple to triple exponential smoothing with additive or multiplicative trend and seasonality and can automatically select a model from this class.

Regressive A generally stronger but more computationally intensive class of models uses regression. Basic linear regression fits a line through the data using least squares and can predict any value between the observed data points or extrapolate beyond the measurement. Can also be used as a simple time series forecasting method. Referenced by one article abstract.

Autoregression is a regression of the time series with the lagged values of itself. It quantifies the so-called self-similarity of a time series in a statistical fashion. An autoregressive (AR) model has a number of coefficients equal to its order. The next value of a time series is forecasted as a sum of the lags multiplied by corresponding coefficients. An AR(3) model therefore takes into account a history of 3 time units. The coefficients are fitted numerically using the least squares method. It models trend and with order ¿ 1 can also have cyclic behavior, although there is a special class of models for more general seasonal series.

Moving average (MA) models extend the simple moving average by measuring errors not only from the last data point but from as many points from the history as the order of the model. The next data point is computed as the sum of these errors multiplied by corresponding coefficients. The estimation of those is more difficult than in case of AR models and requires non-linear least squares or maximum likelihood methods.

The ARMA(p,q) model class contains as additive terms both an AR(p) model and a MA(q) model. This class of models is however sensitive to the stationarity of the time series. As per Box-Jenkins, a method to bring a time series closer to stationarity is to apply differencing, that is to create a new time series containing differences of adjacent members of the original. This is done inside an ARIMA(p,d,q) model, where I stands for integrated and the parameter d specifies the number of differencing steps applied. It can be used for medium term forecasts. The ARIMA method, as a generalization of both AR and MA, is used in [18] and in [20] along with a control theoretic approach to predict batch queue duration. A more mathematical description of ARIMA can be found in this thesis in subsection 3.1.2.

SARIMA(p,d,q)(P,D,Q) is the seasonal version of ARIMA. The first three parameters control the first lags similar to ARIMA. The other three parameters control the number of seasonal lags and differencing. Seasonality is, similarly to Holt-Winters, modeled separately for each point in the seasonal pattern. An ARIMA(P,D,Q) model is fitted for lags that are multiple of the seasonal frequency. This class of models can be used for long term forecasts if the time series is seasonal. There is a function auto.arima() in R which tries to automatically find the right number of lags for a (S)ARIMA model. Its use is, however, rather time-consuming. Referenced in [18].

Hybrid There is also a class of models that combines several approaches. One such is tBATS (trigonometric Exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal components). This model combines seasonality expressed using a Fourier series (which is frequency-domain analysis using trigonometric functions), an ETS model augmented with Box-Cox transform (a generalization of logarith-

mic and power transforms used to bring any non-linear trend to linear, which is capturable by ETS) and works around the limitations of exponential smoothing, which cannot capture self-similarity, by using ARMA on its residuals. It seems to be the state of the art in statistical methods and can be used for long term forecasts. Reportedly takes a long time to fit, but not as long as auto.arima(), whose execution time is also highly dependent on input data [18].

2.1.2.2 Machine learning

Other approaches for time series analysis come from the area of machine learning and can also be used for other kinds of data, for example spatial observations.

Time series analysis One popular class of algorithms that can also be used on time series is simulated neural networks. A neuron is simulated as function with one output and many inputs. The output is a linear combination of its inputs, gated through an activation function, which can be for example a threshold or sigmoid (smoothed threshold). A neural network with three layers can approximate any function, and the most interesting property is that neural networks can automatically learn from errors and update their coefficients at runtime. In the context of time series prediction, the inputs of the first layer of neurons are mapped to successive lagged values of the time series, the second layer is hidden and the output layer of one neuron is connected to the predicted value. The power of this model is the same as that of ARIMA, however neural networks have the ability to adapt to changes in the time series, whereas statistical methods demand stationarity. The drawback is that the mechanism of learning is rather heuristic in nature and the selection of parameters is an optimization problem in itself. It was used in four articles, one of them being [25], where a network with 20 inputs and one output was used for one-step-ahead prediction of a time series with 5 minute frequency to decide when to turn off physical machines in a cloud.

Another class of models that can be used for time series analysis is Markov models. They are backed by Markov chains, which are a comprised of states and transitions. Each transition is dependent only on input and current state, when trained on input data, probabilities can be assigned to each transition. Therefore, from a given state, the most probable future evolution of the system can be predicted. Markov chains are best used if the input space is discretized, so the prediction will also be given in discrete steps. In [31], a generalized version of Markov chains is used that is dependent not only on current state but also on one previous state. It predicts system state in a short forecast horizon for anomaly detection. In [26], a Markov chain is trained not directly on the values of a time series, but on FFT signatures of its frequency domain. It can forecast 10 samples ahead and is used for frequency scaling and VM migration.

Classification Other kinds of prediction than time series forecasting have been found in the studied literature. For example anomaly detectors used machine learning approaches to detect abnormal states. In [31] a naïve Bayes classificator is used to classify states as

normal or abnormal. It is a supervised learning method, meaning that before classification, samples labeled with desired output have to be given to the system by other means. It is also possible to continue teaching the model at run time. In the learning anomaly detectors, abnormal states are identified by monitoring SLO violations and presenting samples from such times as abnormal. The system can then ideally identify the abnormal state ahead of the SLO violation and take corrective action.

The article [30] then uses a self-organizing map to the same ends. The SOM is a non-supervised learning technique; it is a 2-dimensional neural network which learns from incoming samples. The authors gave each sample 8 parameters from the measured system. Each neuron has its coordinates, initially randomly generated. An incoming sample is passed to the neuron with nearest coordinates, which are then modified towards the parameters of the sample by exponential smoothing. The neighboring neurons are also updated, but with lower weight. The result is mapping of the 8-dimensional input space to two dimensions, where similar parameters of the input will be also near in the map. The normal/abnormal classification was done by judging how many times each neuron was updated. The often used neurons represented normal system state, while the less used were outliers.

The article [29] was on profiling grid tasks and predicting their run times on different machines. It employed k-means regression. This is another unsupervised learning technique. First, the k-means algorithm for data clustering was used on the profiled data to find the characteristics of different profiles on different machines, then regression was used to predict the run times of new traces.

2.1.2.3 Pattern matching

Time series analysis can also be done by algorithms inspired by stringology, which has strong tools for non-exact matching of strings. This requires discretization of the values of the time series, so that strings of states may be compared. The recently observed pattern in a time series is searched in its history (with limited lookback), the results are then ordered by the closeness of the match, and the prediction is based on what values came next after the historically observed samples. These algorithms will be very strong on time series that show little autocorrelation, but exhibit repeating patterns of behavior. The article [24] uses such a technique, where the strings are composed of some defined levels of slope from the time series plot. Strings of symbols -4..+4 are obtained by grading the slopes observed during 5 minute measurement intervals. The prediction of the model is used to turn on and off physical hosts in a cloud.

Article [17] uses a modified Knuth-Morris-Pratt algorithm. The original KMP speeds up searching substrings in a text by intelligently skipping comparison of characters that cannot match. It pre-computes a state machine based on the searched word to find out how many characters to safely skip when a mismatch occurs at each position of the compared word. The said modification was to enable non-exact matching and is the topic of the article. The resulting predictor was used with a resolution of 100s and lookback of 100 samples and tested on server load traces.

2.1.2.4 Game theory

One article abstract described a method of resource allocation in overloaded clusters that employed game theory. Each tenant had his performance goals and a penalty of not meeting them and an iterative algorithm was devised to compute an allocation of VMs to PMs in a way that as many goals as possible will be fulfilled and the total penalty will be minimal.

2.1.2.5 Queueing theory models

Queueing theory was originally focused on problems in telecommunication, but can be applied to problems computer science, transportation, or even real life. It constructs models of queueing systems, which have an input flow of service requests (from a finite or infinite amount of users) and a number of service lines that serve these requests. Both the input flow and the service times of the lines are stochastic processes, and the theory can analytically provide answers to questions like what will be the distribution of waiting times of the requests in a queue, or what will be the probability of denial if queueing is not allowed. Inversely, knowing SLOs, one can compute the number and speed of service lines necessary to achieve it with a certain probability.

These basic service system models may be chained together using Markov chains to create queueing network models. They can for example model a situation where a request will be first served at the CPU, then with a certain probability will wait for a hard disk, then return to the CPU queue, and finally go to the network card to be sent out.

Three uses of queueing theory models have been found in literature. Four articles have used it to create a simulation of an application in the cloud after having profiled it. For example, in [21], a simulation of a batch job system with autoscaling is performed. Two articles proposed queueing network models of multi-tier application in the cloud, for example [23] simulated a system with load balancers, and it made autoscaling decision based on the lengths and throughputs of these queues, without having constructed a model by profiling. Two articles used queueing theory for modeling grid workloads. For an overview of modeling problems in grids, a doctoral thesis abstract on the topic [32] is recommended.

2.1.2.6 Control theory

Control theory is a discipline that was originally used in manufacturing, but is also applied in mechanical engineering and computer science. It studies dynamical systems and the means of their control, that is keeping a possibly unstable system in a desired state. Most models have architecture with a feedback loop, where a controller drives the controlled system based on the desired state and the current state of the system based on measurement. The system is described using a mathematical model and the controller uses the model to drive the real system. Model predictive control, referenced in an article abstract, is a controller that predicts several steps ahead based on the current state and tries to bring the system to the desired state using minimal intervention. Article [21] uses a controller to drive a queueing theory model of and autoscaling cloud system.

2.1.2.7 Heuristics

Several articles used a heuristic approach, that is a method derived from common sense and turned into an algorithm. For example an automatic resource allocation scheme can be based on the detection of maximal values from load traces. Article [15] present a combined homeostatic and tendency based predictor. It assigns weights to two observations. First, a time series may return to its average soon after a spike occurred. Second, a time series that had a trend for several past observations may continue to exhibit that trend. Tuning this predictor, the authors outperformed other approaches present in the Network Weather Service.

2.1.3 Summary

In this section, a study of existing scientific articles with the topic performance prediction in cloud computing has been performed. Although the original idea was to summarize time series forecasting methods used in automatic scaling, the study also contains other approaches such as classification, queueing and control theory. Also other research goals than green computing have been found, for example anomaly detection, VM placement optimization, market regulation and performance simulation. While the list of articles is not conclusive, as some authors may have used different key words from those that were searched, the list of goals and methods should be sufficient to give a good overview of the state of the art and help in deciding which research topic to pursue.

Overall, looking at the methods, we can see methods that are best used in one-step-ahead mode. These simpler methods have the advantage of speed, but lack accuracy at longer forecast horizons. They are best used in anomaly detectors, where this horizon is sufficient, or to augment more complex long-range methods, forming an ensemble.

The second class would be methods that capture trend and are good for medium range forecasts or for data that do not have seasonality. Whether this class is or is not sufficient depends on what forecast horizon is required by the application. They should still offer good speed and are thus well suited for situations, where the action can be done frequently based on the prediction, such as with DVFS.

The seasonal models then exploit cyclic behavior in data to provide long term forecasts, meaning several periods of the repetitive pattern. This capability is appealing, but is still remains to be seen if it will be necessary in the context of cloud computing problems. These models were not used in any of the researched articles, but could be interesting in scenarios when, e.g., frequent turning on and off of machines or VMs in a public cloud is costly and thus a longer forecast horizon is needed.

For some special time series, which exhibit repetitive behavior, which is not exactly seasonal, pattern matching approaches will probably provide the best predictions. However, it will be more difficult to obtain confidence intervals as these models are not backed by statistics.

The strength of machine learning approaches is hard to judge. Neural networks used directly on time series have the same power as ARIMA models, however the process of

model fitting is vastly different and it will depend on the input data, which will perform better. Likely, they will perform better on series that are quickly changing and thus violating the principle of stationarity required by the autoregressive models, which in that case need short learning history and frequent re-training.

2.2 Autoscaling Simulation

This section attempts to be an exhaustive study of available possibilities to implement a simulation of a cloud autoscaler. It starts with the most specific tools, which are focused on the simulation of clouds and data centers, continues with more generic tools for performance analysis, and ends in a list of simulation languages for discrete event simulation, which would have to be used when none of the previous possibilities was found to offer the needed features and flexibility.

2.2.1 CloudSim and other cloud simulators

During an earlier state-of-the-art analysis of cloud computing in general, CloudSim was found to be cited by several articles dealing with optimal resource allocation in clouds. One is written by Beloglazov, Abawajy, and Buyya [33], the authors of CloudSim, and deals with optimizing data center power usage through heuristic VM placement. Jeyarani, Nagaveni, and Ram [34] approached the same problem using particle swarm optimization algorithms. Both these articles work with SLA (Service Level Agreement) defined as a certain amount of MIPS (Millions Instructions Per Second) of processing power and focus on static allocation of VMs and their migration, whereas we attempt to use the simulator to calculate actual latencies of web application requests and need to change the number of VMs at runtime.

De Oliveira, Ogasawara, Ocana, Baiao, and Mattoso [35] used CloudSim to prove an adaptive scheduling strategy for the SciCumulus workflow engine; however, details of the simulation are unclear from the paper.

Zhao, Peng, Xie, and Dai [36] present a review of available simulation tools for Cloud Computing. Contained in the survey are CloudSim, CloudAnalyst, SPECI, GreenCloud, GroudSim, NetworkCloudSim, EMUSIM, DCSim, iCanCloud, and some real testbed clouds. All these simulators are based on the discrete event model. GroudSim focuses on simulation of batch computations; SPECI and DCSim seem to be focused on data center simulation and will probably be unsuitable for autoscaled application simulation. GreenCloud is based on the NS2 packet-level network simulator and could potentially have high accuracy. iCanCloud is the most recent of the projects and is actively developed, but will also likely lack the functionality we need. Our questions sent to the authors about the matter have not been answered. The other projects are related to CloudSim.

Going beyond the survey looking for articles on CloudSim accuracy, one can find the introduction of the simulator SimIC [37] for interclouds, which builds upon the same sim-

ulation event engine as CloudSim and compares its precision to CloudSim, trusting its design to be the etalon of accuracy.

The IBM technical report [38] on the other hand states: "CloudSim does not meet our requirements in terms of scalability, accuracy (especially of the hardware modelling) and modularity". They have developed their own simulator and tested it on infrastructure operation workflows in OpenStack. They dismiss DCSim, GreenCloud and SPECI as too specialized and iCanCloud as lacking queueing for software resources.

The newest project based on CloudSim is probably CDOSim [39], where CDO stands for Cloud Deployment Option. According to the article, it enhances the user-centric features of CloudSim. It includes the ability to start and stop VMs at runtime, an auto-scaling function, a CPU utilization model, allows application architecture and performance model to be imported and allows multi-tier application to be modeled. Most notably, it is the only other paper found, which includes an accuracy test of the CloudSim engine. The test is done on a Java web application under automatic scaling and the authors report relative error of 30.64% on CPU utilization and 37.57% on latency, averaged over a daily traffic pattern. The authors state that the highest error was observed during a traffic spike. Perhaps the simulation engine cannot reliably model overload situations. The problem with CDOSim is that it was only used as a component of CloudMig Xpress, which seems to be discontinued. Source code is not freely available and the compiled Java application does not allow direct access to CDOSim functions, so no verification is possible.

2.2.2 Analytical solvers for queueing network and other formal models

Another family of simulators that could be useful is Queueing Network Simulators. The advantage is that these models have firm theoretical grounds, the system to be solved is described declaratively, not procedurally as in discrete event simulators, and the user is less constrained by the assumptions of the simulation environment's authors. The downside is that the models employed describe steady state only. Therefore, to simulate an autoscaled application, we would have to find all the steady states the system goes through and run the simulation for each of them. Transitional effects of adding or removing VMs at runtime would not be captured. Also, the simpler models expect exponentially distributed interarrival and service times, which can cause them to diverge from reality if the traffic pattern is very bursty.

The starting point for a list of available QN solvers were chapters 11 and 12 of the book [12], which also contains a comprehensive study of all formulas for solving QNs, Markov Chains, and Stochastic Petri Networks. The list can be categorized by the formalism used in the simulator or solver. We also went through links in articles describing the software pieces and public link collections such as the one by Hlynka[1] to find out which of the tools are still being developed or at least made available.

The first category is software that introduces its own simulation language that is not

[1] `http://web2.uwindsor.ca/math/hlynka/qsoft.html`

based on any formal language. Among those are languages developed by Alan Pritsker[1] – Q-Gert (1977) and SLAM II (1986), and the last one, Visual SLAM/AweSim (1997). These had applications in the army, and in industrial and commercial modeling. The languages had some properties of queueing theory, but the simulations were event-based and supported discrete and continuous state space. Currently, there is a project and a company called AweSim at the Ohio Supercomputing Center[2] specializing in industrial simulation, though we could not find a connection to Alan Pritsker on their site.

GPSS (1965) is a modeling language with properties of QN and Petri nets, meaning that it can simulate queues and, additionally, model contention and locking of other resources. Therefore, it must be interpreted by discrete event simulation. It was designed for manufacturing but had broader application. The original implementation was by IBM; now there are at least three commercial implementations and one for academic purposes[3] [40].

Moving forward to simulation languages that build on the Queueing Network formalism, the first we could find is RESQ (1977), which was developed at IBM. It was probably used through the 90s, but we could not find a current version of it.

QNAP2 (1984) was a queueing network simulator developed at INRIA. It was sold to the spin-off Simulog and used as part of its product MODLINE[4]. Simulog was itself sold more than one time[5] and no current version of QNAP2 or MODLINE was found.

The system with the highest number of solution methods to date is PEPSY-QNS (1994)[6]. It claims to have 50 methods implemented, probably covering all algorithms described in the book [12], one of them being discrete event simulation. Most notably, it can model non product-form QNs with arrival and service time distributions specified by mean and variance. There was a command-line version and an X-Window interface, which do not seem to be available anymore. The WinPEPSY version [41], although it does not contain all the algorithms from the Unix version, is still available[7]. However, it is focused on GUI and is not likely to be usable from the command line.

PDQ (2000, last version 2013)[8] is described in the book [42]. It is written in C but provides integrations for other programming languages, namely Perl, Python, R, Java and (unmaintained) PHP. Being a library, it has no user interface itself, but its strong point is the ease of integration into other projects. It is capable of solving open and closed single or multi-class product-form queueing networks. It only contains three algorithms and is limited to exponential arrival and service distributions. However, from our experience, that is enough for most purposes. We are using this library in our autoscaling simulator in R and have integrated it into a web tool for cloud capacity planning.

[1] https://en.wikipedia.org/wiki/Alan_Pritsker#Discrete-continuous_system_simulation
[2] https://www.osc.edu/content/the_awesim_advantage
[3] http://jgpss.liam.upc.edu/about
[4] http://www.ercim.eu/publication/Ercim_News/enw24/simulog.html
[5] http://en.wikipedia.org/wiki/Esterel_Technologies
[6] https://www4.cs.fau.de/Projects/PEPSY/en/pepsy.html
[7] http://www7old.informatik.uni-erlangen.de/~prbazan/pepsy/
[8] http://www.perfdynamics.com/Tools/PDQcode.html

JMT (2006, last version 2013)[1] [43] was developed at Politecnico di Milano and Imperial College London and is a collection of tools written in Java for simulation and analytical solving of queueing networks. It contains six tools, namely JSIMGraph for graphical model specification, JSIMWiz and JMVA for textual model specification, where the second interface is constrained by the possibilities of analytical solvers. There are nine algorithms available. Using both simulation and analytical approaches, a wide variety of model types are specifiable, including non-exponential arrival and service times, load-dependent servers, fork-join operations, various load-balancing strategies. Further, there is JABA for bounds analysis, JMCH for Markov-chain visualization and JWAT for logfile analysis. The tools also have a command-line interface, which accepts models in XML files, and, due to the GPL license, could lend themselves to integration into other Java projects. From personal experience, we must say that their strongest point is the graphical interface, which includes what-if analysis to vary parameters and result graphing, making it excellent for teaching or quick manual model evaluations.

queueing[2], originally qnetworks [44] (also still listed as such in Hlynka's links), is a plugin for GNU Octave that implements algorithms for the analysis of Markov chains, classical single-station queueing systems and analytical solving of single or multi-class open or closed product-form queueing networks. It has the ambition to cover the book [12] and is in active development.

Another formalism used in performance engineering is Stochastic Petri Networks. They have higher expressive power at the cost of lower intuitiveness. SPNs can be used to model inter-process blocking, simultaneous resource possession, and can capture transient events, not only steady state. This makes them suitable for reliability analysis as well as for modeling performance engineering scenarios. A well-known package for SPN modeling is SPNP (1989) [45] from Duke University[3]. The model input is in a special C-like language CSPL. Markovian SPNs can be analytically solved, and others are simulated. The downside of this tool that it has an academic-only license.

From mixed-model tools, we mention SHARPE (2000) [46] from the same team and with the same license, which is a multi-model simulation tool that can process Markov chains, Reliability Block Diagrams, Fault Trees, Queueing Networks, and Petri Nets. It has a simpler language and features a GUI.

MOSEL-2 (1995) [47] from Universität Erlangen-Nürnberg[4] defines its own language that is based on queueing networks and translates it to input for SPNP and other tools. Based on model parameters, either analytical methods or simulation is used to solve the model.

A new formal specification for performance engineering was found when examining the Palladio Component Model (2007) [48] from Karlsruhe Institute of Technology[5]. PCM is a modification of the Eclipse development platform targeted at software architecture

[1] `http://jmt.sourceforge.net/`

[2] `http://www.moreno.marzolla.name/software/queueing/`

[3] `http://people.ee.duke.edu/~kst/software_packages.html`

[4] `https://www4.cs.fau.de/Projects/MOSEL/`

[5] `https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model/Documentation,_Tutorials,_and_Screencasts`

modeling, including deployment diagrams. The models can then be evaluated in four dimensions - performance, reliability, maintainability and costs. For analysis, the models can either be simulated by the discrete event approach, executed on the target platform in the form of a performance prototype generated by the ProtoCom module, or using the PCM2LQN module converted to a Layered Queueing Network.

LQNs were developed at Carleton University[1] as an extension of QNs [49]. They move the expressive power of QNs closer to SPNs, while having better readability. They model simultaneous resource possession in systems that can be split into layers. E.g., by modeling contention for software resources that are constrained by hardware resources and need to wait for external services. Modeling of post-reply processing and reliability predictions are also possible. The downside is that the original LQNS package has a restrictive license. There is also the LINE solver [50] that accepts the same input format and is distributed under GPL[2].

Reading the documentation of PCM, the SimuLizar plugin [51] was found[3]. It is an extension to simulate self-adaptive systems, which are in the software engineering world an equivalent to our cloud computing term autoscaler. They present a fuzzy temporal logic language to express relaxed non-functional constraints and extend PCM to adapt the model when the constraints are not met. The evaluation section assumes a system which contains one private server and one rented server, which is paid by fine CPU time. This is not very realistic in the cloud world. We only know one provider, which works this way. It is a local cloud provider[4] who uses container-based virtualization for their service and bills by CPU-cycles, not hours. The authors probably omitted a scale-down rule, so the provider bill with more frequent adaptations was higher than with less frequent. An autoscaler with a higher granularity should be faster not only to allocate resources but also to release them, which does not occur in the shown traces. Without knowing the input time series that drove the simulation, it is not possible to say this with certainty.

2.2.3 Languages for discrete event simulation

When no existing simulators for cloud computing or general performance model solvers would fit a particular purpose, one could write his or her own simulation using some existing discrete event simulation framework. The starting point for the survey is the book Queueing Networks and Markov Chains [12].

The first such package was probably SimScript (1963), at that time written in Fortran. The current version III [52] is commercial[5] and is an object-oriented 64bit language with graphical functions.

GASP (1964) was also written in Fortran and was being developed by Alan Pritsker. The last version is likely GASP-V from 1979.

[1] http://www.sce.carleton.ca/rads/lqns/lqn-documentation/

[2] https://code.google.com/p/line/

[3] https://sdqweb.ipd.kit.edu/wiki/SimuLizar

[4] 4smart.cz

[5] http://www.simscript.com/products/products.html

CSIM (1985) is a simulation library for the C language. The current version 20 [53] is commercial [1].

MATLAB (1984) is a mathematical programming suite. It also contains Simulink for various simulations including the SimEvents package for discrete event simulation [2]. It is a commercial product.

OPNET (1986) is an discrete event simulator for communication networks. It was commercial before, but most recently is has been bought in 2012 by Riverbed and rebranded SteelCentral NetModeler Suite [3].

ns is an open-source project with the first version ns-1 from (1995). The current version ns-3 (2008) [54] is written in C and Python and is in active development [4]. It is focused on network and wireless protocol simulation. It is not compatible with previous versions and has no GUI.

OMNeT++ (1999)[5] [55] is another open-source discrete event simulator for communication networks that is in active development. It is written in C++ and has a GUI based on Eclipse as well as a command-line interface accepting input in the NED simulation language. There is a manual on queueing simulation in OMNeT++ [6].

SimJava (1996) [56] from the University of Edinburgh is a low-level discrete event simulation engine in Java. It was the base of GridSim and earlier versions of CloudSim. The last version is from 2004.

SimPy (2002) [57] is a low-level discrete event simulation engine in Python[7]. It is an open-source project in active development.

[1] http://www.mesquite.com/products/csim20.htm
[2] http://www.mathworks.com/discovery/queuing-theory.html
[3] http://www.riverbed.com/products/performance-management-control/opnet.html
[4] https://www.nsnam.org/
[5] http://www.omnetpp.org/
[6] http://omnetpp.org/doc/queueing-tutorial.pdf
[7] http://simpy.readthedocs.org/en/latest/

Chapter 3

Overview of Our Approach

3.1 Interactive Workload Prediction

While the theoretical advantages of cloud computing are widely known – private clouds build on the foundations of virtualization technology and add automation, which should result in savings on administration while improving availability. They provide elasticity, which means that an application deployed to the cloud can dynamically change the amount of resources it uses. Another connected term is agility, meaning that the infrastructure can be used for multiple purposes depending on current needs. Lastly, the cloud should provide self-service, so that the customer can provision his infrastructure at will, and pay-per-use, so he will pay exactly for what he consumed.

A private cloud can be used for multiple tasks, which all draw resources from a common pool. This heterogenous load can basically be broken down into two parts, interactive processes and batch processes. An example of the first are web applications, which are probably the major way of interactive remote computer use nowadays, the second could be related to scientific computations or, in the corporate world, data mining.

This division was chosen because of different service level measures used in both the fields. While web servers need to be running all the time and have response times in seconds, in batch job scheduling, the task deadlines are generally in units ranging from tens of minutes to days. This allows a much higher amount of flexibility in allocating resources to these kinds of workloads. In other words, while resources for interactive workloads need to always be provisioned in at least the amount required by the offered load, a job scheduler can decide on when and where to run tasks that are in its queue.

When building a data center, which of course includes private clouds, the investor will probably want to ensure that it is utilized as much as possible. The private cloud can help achieve that, but not when the entire load is interactive. This is due to the fact that interactive load depends on user activity, which varies throughout the day, as seen in Figure 1.1 in Section 1.4.4.

In our opinion, the only way to increase the utilization of a private cloud is to introduce non-interactive tasks that will fill in the white parts of the graph, i.e., capacity left unused

by interactive traffic (which of course needs to have priority over batch jobs).

The goal is that the scheduler will be fed with data about the likely amount of free resources left on the cluster by interactive processes several hours into the future by a predictor. This will ensure that the cluster is always fully loaded, but the interactive load is never starved for resources. It is also the reason why we have investigated the two long-range seasonal forecasting methods described below.

With a predictor, instead of seeing only the current amount of free resources in the cloud, the batch job scheduler could be able to ask: "May I allocate 10 large instances to a parallel job for the next 4 hours with 80% probability of it not being killed?"

Prediction of load or any other quantity in time is studied in a branch of statistics called Time Series Analysis and Forecasting. This discipline has also been studied as part of this project and the results are presented in this section.

A good tutorial on Time Series Analysis is written by Keogh [58]. It has very wide coverage, mainly on filtering, similarity measures, Dynamic Time Warping and lower bounds on similarity. However, the solution was found elsewhere, although clustering on particular days and offering the next day after the best match as forecast is also a valid approach and was evaluated as better than the two others presented here in the bachelor thesis of Babka [59] on photovoltaic power plant output prediction.

3.1.1 Holt-Winters exponential smoothing

Due to the fact that did not have real data from an autoscaled cloud environment, it was decided to obtain experimental data from single servers of a web hosting company. These are monitored by Collectd and time series data stored in RRDTool's Round Robin Databases. While examining the documentation for export possibilities, a function by Brutlag [60] was discovered, which uses Holt-Winters exponential smoothing to predict the time series one step ahead and then raise an alarm if the real value is too different from the prediction. This allows to automatically detect spikes in server of network activity.

A good description of exponential smoothing methods including mathematical notation is written by Kalekar [61]. In this document, a summary is presented in the Paragraph 2.1.2.1

Estimation of the parameters can either be done by hand and evaluated using MSE (Mean Squared Error) or MAPE (Mean Average Percentage Error) on the training data (a quick explanation of their significance is in Hyndman [62] or in this document in Subsection 4.1.3), or it can be left to statistics software, which can do fitting by least square error. For the experiments in this work, the R statistics package [63] was used, particularly the forecast package by Hyndman [64]. The RRDTool implementation is not suitable as it only forecasts one point into the future for spike detection.

An introduction to time series in R, including loading of data, creating time series objects, extracting subsets, performing lags and differences, fitting linear models, and using the zoo library is written by Lundholm [65]. A summary of all available time series functions is in the time series task view [66], while a more mathematical view of the

capabilities including citations of the authors of particular packages is in McLeod, Yu and Mahdi [67].

3.1.2 Box-Jenkins / ARIMA models

ARIMA (Autoregressive, Integrated, Moving average) models are intrinsically based on autocorrelation. They seem to be the state of the art in time series modeling and are a standard in economic prediction (e.g., [68] is a textbook for business schools and MBA).

Neural network methods were also studied, but, as Crone's presentation, which is also a good source on time series decomposition and ARIMA [69], suggests, their forecasting power is equal to ARIMA, only the fitting method is different. It may be more powerful in that it is non-linear and adaptive, but has many degrees of freedom in settings and the result is not interpretable.

As per the NIST Engineering Statistics Handbook [70, chapter 6.4.4.4], which is a good practical source on all methods discussed here, the autoregressive and moving average models were known before, but Box and Jenkins have combined them together and created a methodology for their use.

There are three major steps in the methodology: model selection based mainly on the examination of autocorrelograms (ACF) and partial autocorrelograms (PACF), then model estimation, which uses non-linear least square fitting and/or maximum likelihood and is best left to statistical software, and lastly model validation, which uses ACF and PACF of residuals and the Ljung-Box test.

An autoregressive (AR) model computes the next data point X_t as a linear combination of previous ones, where the number of lagged values considered is determined by the order p of the model.

$$X_t = c + \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t$$

The parameters are the constant mean c and the coefficients φ of each lag. They can be computed by linear least squares fitting. A model of order greater than one with some coefficients negative can exhibit cyclic behavior. The term ε represents the residuals of the model, which should ideally be white noise.

A moving average (MA) model works further with the errors. The next data point is a linear combination of differences of past lags from the moving average μ, where the number of lags considered is the order q of the model.

$$X_t = \mu + \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}$$

Again, each term has a parameter θ that needs to be estimated. The estimation is more difficult as the errors cannot be known before the model exists, which calls for an iterative non-linear fitting procedure.

When both of these models are used together, an ARMA model is obtained:

$$X_t = c + \varepsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}$$

The I in ARIMA stands for integrated, which represents the inverse operation to differencing, which is simply creating a new time series from the differences between the values of the original one:

$$X'_t = X_t - X_{t-1}$$

As the AR and MA models assume that the time series is stationary, meaning that it has stable location and variance, the difference operator can often be used to transform a series to stationary. Sometimes, the operation has to be performed more than once, d times, to coerce the time series to stationarity. The model is fitted to the transformed series and an inverse transform is used on the resulting forecast. Other useful transformations are logarithms and power transforms, which may help if the variance depends on the level. They are both covered by the Box-Cox transform (see [68, chapter 2/4]).

3.2 Simulation of Cloud Autoscaling

The lack of large scale data and experimental environments with live traffic led us to focus our project on the client side of IaaS (Infrastructure as a Service), instead of the provider side and Green Computing, or the maximization of utilization in private clouds through heterogeneous load proposed in our earlier articles, and to develop the hereby presented simulation method, with the goal to alleviate fears associated with automatic scaling. Even if there is data, the proposed forecasting algorithms from our previous work need to be tested and tuned in a simulation environment before being implemented in a real scaling application.

As a widely referenced work (See Section 2.2.1), the CloudSim event-based simulator was the first candidate for a simulation platform. One part of our work focused on implementing a reactive autoscaling policy inside of CloudSim, which required us to write methods to add and remove VMs at run time and to provide improved statistics gathering. We have observed poor accuracy in load test experiments (Section 3.2.2), and therefore did not implement the predictive autoscaler in the simulator, but instead a) redirected our efforts to fixing CloudSim and b) creating a suitable replacement simulation platform.

After performing a load test, we suspected that the problems in the simulator were due to erroneous queueing logic. We have rewritten that, and also the traffic generation code, getting a version of CloudSim that is capable of reproducing a load test experiment with reasonable accuracy and in a manner consistent with queueing theory.

It is necessary to state in advance that the difficulties we have had with CloudSim are mainly due to the fact that it was not primarily designed to simulate interactive traffic. As seen from the examples on its website, it is intended to simulate batch computations on virtualized infrastructure. It is mostly used with Cloudlet durations in hundreds of seconds,

while we need to work in millisecond ranges with submission rates in tens to thousands of jobs per second, which is why the described errors have surfaced. The current version of the main distribution is 3. The CloudAnalyst distribution is actually designed to simulate world-scale interactive services and implements some features such as higher time resolution and a traffic generator to that end. It is based on CloudSim version 1 beta. It should not be used either for that nor for any other purpose unless our modifications are applied.

We believe that the latest CloudSim could also benefit from an audit of its queueing code. However, it should be fit for its purpose of simulating VM placement strategies, datacenter power optimizations and cooperation of compute and networking. However, we find it even less suitable for autoscaling experiments on interactive traffic than the older CloudAnalyst, because it operates on a larger time scale. Also, important functions for latency and throughput measurement have been removed when the simulation core SimJava was replaced by a custom one.

Due to the difficulties with CloudSim, we have decided to write our own simulator for automatic scaling and to base it on a queueing network model. The implementation is written in the R language and uses the PDQ open-source queueing network analyzer library by Gunther [42] as its engine. The use of R also allows for a much simpler integration of forecasting methods than Java and leverages R's graphing capabilities. Several other modeling tools were also studied (See Section 2.2.2). This queue-theoretical model is also found to be consistent with the practical load test experiment, even better than the modified version of CloudSim we created for the purpose [71]. It is also several orders of magnitude faster. We show the possibilities of the model by simulating several threshold-based autoscaling algorithms on a trace from an e-commerce website under fluctuating load before and during Christmas holidays. By doing this evaluation, we have found that using a combination of latency and utilization or latency and queue length as autoscaling metrics may be better than using any single one.

3.2.1 CloudSim and Changes to implement Autoscaling

As seen in the previous section, CloudSim [2] is a well-known simulator in the cloud computing research community. It is developed at the University of Melbourne and is based on their previous work, GridSim. It was decided to install this simulator and test it on a threshold-based autoscaling algorithm, before moving to predictive scaling experiments.

The simulator is rather complex. It builds on the event-based simulation framework SimJava, which provides message scheduling and statistics, and has some relics from GridSim (like an application being called Cloudlet, which is a class extending Gridlet). Documentation is rather scarce, apart from journal articles and function headers inside the source code.

The layered structure of the project proved to be a hindrance to its usage. While the framework expects the user to override or modify certain classes of CloudSim to implement desired functionality, most of the classes are inheriting from their counterparts in GridSim, which is only included as a pre-compiled .jar package. Therefore, it is not easy to understand their functionality without downloading the GridSim project and browsing

its code on the side. The numbers and tags of events sent between entities are also undocumented and their declaration is spread between the two projects. Understanding of statistics collection then requires the user to go further, into SimJava documentation.

Concretely, it took us an inordinate amount of time to locate the place in the code where the latency of a request is computed. The articles about the project do not go nearly deep enough to guide the user in how to modify the simulator to his or her needs and what are the open possibilities. When there was a problem with wrong messages being passed, the authors did not comment, neither did the discussion group.

To dampen the learning curve, the CloudAnalyst [72] package was used. It provides a graphical interface to CloudSim and helps set the basic entities of User Bases and Data Centers and their parameters, such as the number of hosts and VMs, balancing policies, computational requirements of Cloudlets, amount of user requests, etc. It was originally developed to simulate worldwide deployments of social network sites. Once it is set up, the software allows to set a workload defined as the number of requests per hour of the users of each User Base, different for peak hours and off peak hours in each region. Once the simulation is finished, it shows some statistics such as the response times seen at each Datacenter and User Base, and the estimated cost for using the simulated cloud system.

Please note once more that the described version of CloudSim here is 1 beta. The latest version has been rewritten to no longer reference GridSim classes. It also dropped SimJava and does event handling on its own. The problem with the lack of developer documentation and unhelpfulness of the discussion group persists. However, the main distribution of CloudSim lacks some important features for autoscaling simulation, such as the UserBase, which actually is a load generator. The main distribution expects the user to specify the submission times and service demands of all Cloudlets in advance, which is not possible for interactive services, where the input flow is specified in statistical terms.

An unexpected surprise was that CloudSim does not provide a function to add or remove VMs while the simulation is running, which is necessary to the implementation of an autoscaler. Normally, all the simulation entities are instantiated at the beginning and terminated at the end of the simulation, see Figure 5 in [1]. It was also necessary to trace the exchange of messages at VM creation and deletion and write functions that inject these messages at run time, even the counters for VM usage in terms of time and money paid to the cloud provider present in CloudAnalyst do not take VM deletion in account. That was fixed as well.

An alternative to implementing VM addition and deletion would be to employ the migration function of CloudSim to move pre-created VMs from a pool that is not serving requests to an active one, such as done by Bessis, Sotiriadis, Xhafa, and Asimakopoulou in [73].

One more problem was with the granularity of offered load levels settings and statistics collection. CloudAnalyst can only specify two load levels per User Base. This is not ideal for an autoscaling experiment. Therefore, a function that provides load levels from an hourly series was created. For some reason, the simulation always stops after one day, even if the simulation length is set to a longer time, so even 24 values may prove to be too few to induce a reasonable number of dynamic changes of the number of VMs.

In addition, statistics collection in CloudAnalyst is done on an hourly basis, which does not allow to capture the transitional effect on VM addition/deletion. However, the SimJava class Sim_stat [74] provides several event counters that can count latencies and rates and compute simple statistical functions on them while the simulation is running. A latency recorder with 5-minute granularity was implemented using this functionality.

With these modifications in place, a simple threshold-based autoscaler reacting to latency (Google App Engine style) has been implemented. Being able to measure average latency and request throughput, it is also possible to compute data center utilization from the Utilization Law and implement a load-based autoscaler (Amazon Web Services style).

Unfortunately, the newest CloudSim dropped the SimJava engine and does not offer a replacement for its statistics collection functions, making our changes difficult to port to it without modifying the new event passing engine. What is available is the submit and finish time for every Cloudlet. SimJava offered to place a counter or time interval collector at any place. Therefore, CloudAnalyst can measure separately the service time at the DataCenter or the perceived response time at the UserBase. Throughputs at various places could be captured by just placing a counter. Counters can be filtered by time intervals when being read, allowing to create time plots easily.

3.2.2 Verification of CloudSim using Load Testing

During the experiments with CloudSim, which led to the work presented in article [71], a question has arisen about the accuracy of the simulation. It was decided to compare a small scale load testing experiment with the simulator. The design of the experiment was chosen to verify the possibility of autoscaling simulation. It creates a constant load on a load-balanced website running on a number of VMS. The VMs are being removed at regular intervals during the experiment. Removal was used instead of addition so that effects like VM start-up time, performance impact on the hypervisor, and cold caches do not affect the outcome. These effects are not simulated in CloudSim or the presented solution. By proving the precision of the model on this simple example, it should be possible to reason about the accuracy of more complex scenarios.

The load testing experiment measures the latency of a web application running in a small cloud. The offered load by the User Base is constant, while the number of VMs serving the application is decreasing from the maximum number down to one. One VM is removed every 15 minutes; the total experiment duration is 2 hours. The load is chosen to be the maximum that can be handled by one instance.

The maximum number of VMs was set to 8 with 1 virtual CPU each, as the available hardware for the experiment consisted of one server, virtualized with Citrix XCP (Xen Cloud Platform) version 1.6, having a 4-core 8-thread "Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz" processor, 32 GB of DDR3 RAM and a 120 GB SSD hard drive. The web application used was a basic installation of Wordpress, which has been used earlier for the evaluation of ScaleGuru.

The Tsung load generator was used for the experiment because of its modern design and features. It is highly scalable due to its use of the Erlang language, capable of distributed

testing, can simulate an open or closed performance model, the load can be scripted in terms of both scenarios and the load curve, and finally, it has good graphical reporting. It was installed on a separate 2-core machine connected to the XCP server through a 100 Mb/s switch.

Between the load generator and the Apache + mod_php web servers, an Nginx load balancer was placed. It had a request queue of 1000 slots and was set to automatically detect removed web server instances. The queue at the Apache server has 511 slots by default. The MySQL database used by Wordpress was installed on each of the 8 virtual machines for simplicity. It is not possible to simulate multi-tier applications in CloudSim, at least not easily. It could be possible rewrite queueing logic to pause a cloudlet at some point, submit another and wait for a message of its finish to unpause.

Beside the reporting capabilities of Tsung, additional instrumentation was created around the application stack. The Collectd monitoring system was used for data collection. Its standard plugins cover raw hardware statistics, as well as Apache, Nginx and MySQL internals. Custom added were observations of latency at all 3 tiers if the stack. The database was monitored on the transport layer using the tcprstat utility; the web server and load balancer were set to log latencies to a file, which was read by collectd. The latency logs captured every transaction, while the throughputs and concurrencies were sampled from server status outputs. The sampling interval was set to 30 seconds and data was routed to a central server.

As the first step in the scaling experiment, a load test was run against one server with default configuration, with a stair-step pattern, until overload. It was found out that for the PHP application without any acceleration or caching, the CPU is the bottleneck. To prevent server crashes, the number of concurrent processes was limited to 20, which was obtained as the total amount of memory per VM (360 MB) divided by the memory consumption of one process, using the formula

$$N_{Processes} = \frac{M_{Total}}{M_{Process}}$$

A further test against the tuned server showed that at 100% utilization, the server could handle 9 requests per second.

The virtual machine was then cloned 7 times and all resulting instances added to the load balancer. Due to the server not having 8 physical CPU cores and needing to handle internal network communications and other hypervisor overhead, the maximum capacity was only around 40 req/s. However, that did not matter in the final experiment, which was focused on latency and ran with a constant load of 9 req/s. Figures 3.1 and 3.2 illustrate the nature of the load test. The load balancer was seeing an average load of 9 req/s as drawn from an exponential distribution by the load generator. The load at the web server was increasing as the other servers were being shut down, until it received the whole load.

In the second step, the experiment was recreated as closely as possible in CloudAnalyst. One User Base was created with 1012 users, each producing 32 requests per hour. There was one Datacenter with one 8-processor machine and 8 VMs. The Internet between their

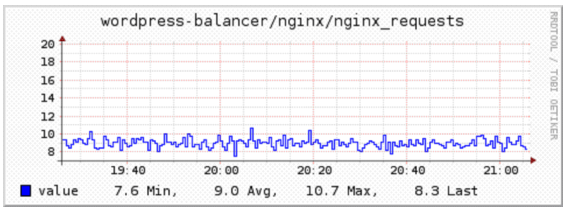


Figure 3.1: The constant incoming load at the load balancer

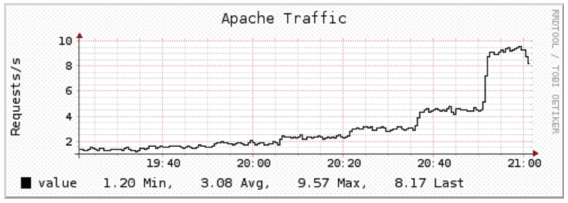


Figure 3.2: Load as seen by the last web server (the one not shut down)

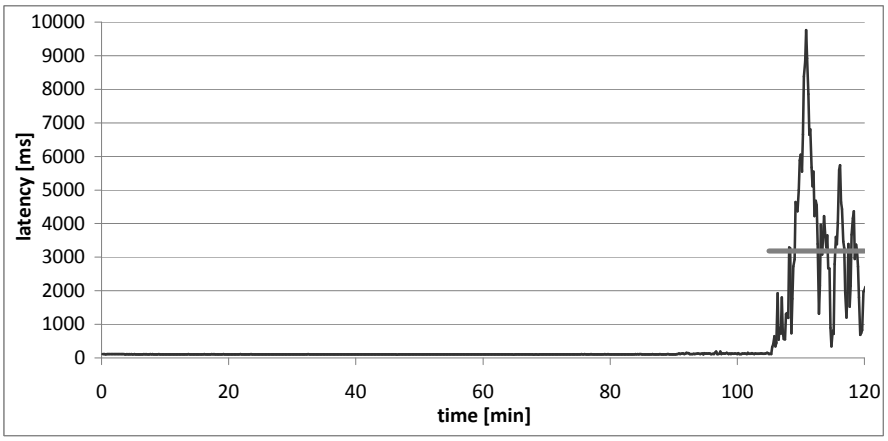


Figure 3.3: Latency at the Tsung load generator, last step average highlighted

locations was set to have a bandwidth of 100 Mb/s and latency of 0.25 ms. One request from the User Base had 300 B and response 7 kB, as measured during the experiment.

The most difficult part was to estimate the computational complexity, which CloudSim expects in millions of instructions (MI), while the machines have power specified in MIPS. From real measurements, only the service demand of a request could be obtained (as minimal latency measured at the web server, including the work done by the database server). The BogoMIPS value of the CPU (7007 MIPS) was taken and, multiplied by the minimal latency of 96 ms, the complexity of 672 MI was input. Interestingly, the resulting minimal latency was too low, so the CPU power had to be decreased to 5500 MIPS for the simulation to match the experiment at least in the first steps.

3.2.2.1 Analysis of discrepancies

As apparent from Figures 3.3 and 3.4, the simulation result is quite different from the real world measurement. The load testing experiment gave a rather stable latency of around 105 ms. The higher values of the first step should be disregarded, as the CPUs of the virtualization server were shared at that point, while the simulation did not consider that. The last three steps had average latencies of 106, 127, and 3181 ms. The last step shows

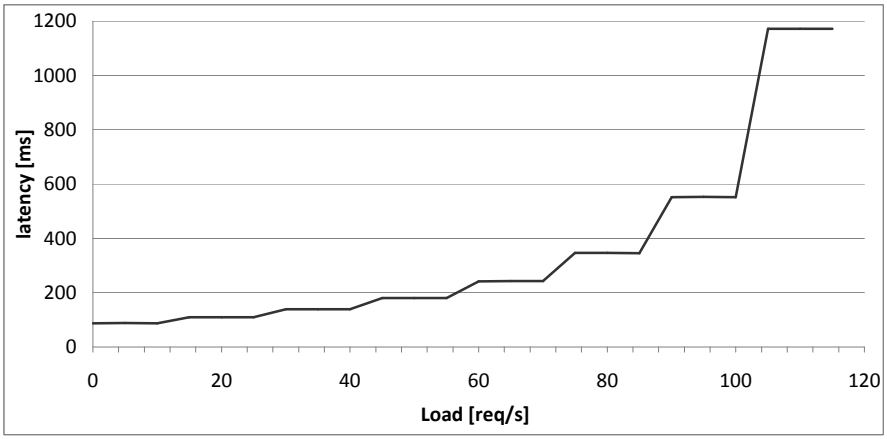


Figure 3.4: Latency results from CloudSim

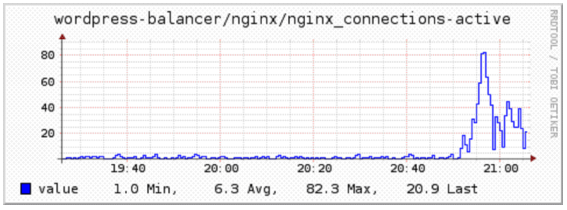


Figure 3.5: Concurrency at the load balancer

that a queue has built at the load balancer, because the remaining web server ran at 100% utilization. The state of the system was still steady, though, and the queue length was dropping after the web server warmed up all threads (see Figure 3.5). With any more load, the queue would have filled all slots, and then the load balancer would start dropping connections.

The resulting latency from the simulator, on the other hand, is increasing every step, proportionally to the load on the web servers. The values in milliseconds are: 89, 106, 138, 178, 267, 332, 581, and 1193. The relative errors are therefore: -19%, 4%, 32%, 73%, 133%, 225%, 333%, and 172%. They were calculated as:

$$\frac{t_{sim} - t_{test}}{t_{test}}$$

The quick conclusion from this is that the simulator diverges from reality by several times and does not predict overload situations.

The only possible conclusion from this test is that there must be significant difference between the messaging and queueing logic of the real setup and the simulator, and it should be the reality that the simulator should attempt to emulate. From Figure 3.6, it is apparent that the concurrency at the web server was mostly 0 or 1, except at the end of the test run. Therefore, there is no cause for its latency to be proportional to the load as in the simulation.

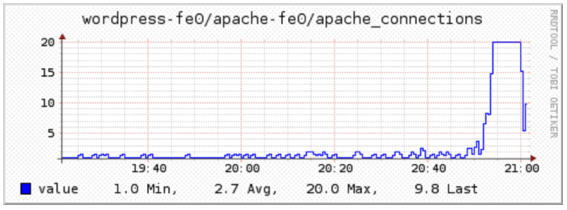


Figure 3.6: Concurrency at the last web server (capped at 20 to prevent swapping)

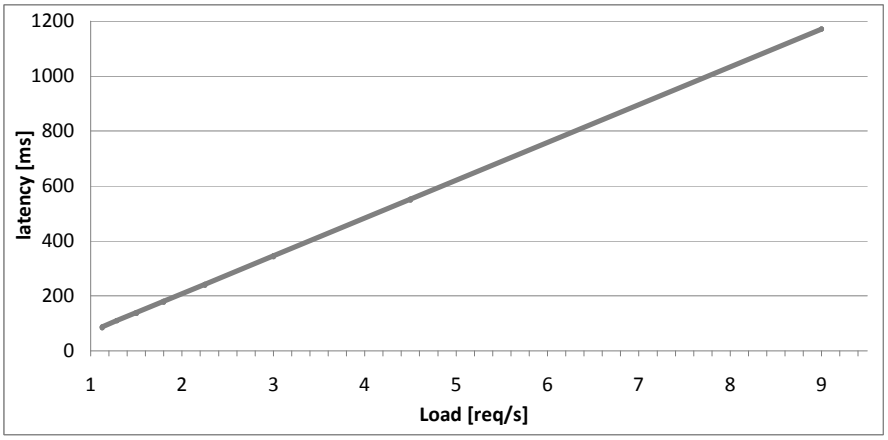


Figure 3.7: Result from CloudSim plotted against load

Plotting the latency results from CloudSim against incoming load instead of time, as seen in Figure 3.7, shows that, indeed, it simulates an overloaded system. When the same experiment is set up in R using the PDQ library for queue network solving, a result close to that from the load test is obtained, see Figure 3.8.

We have identified the problems preventing CloudSim from working on interactive traffic and fixed them, as described in Section 4.2.2. The work was previously published in the article [71].

3.3 Custom Simulator Design based on Queueing Theory

Due to the difficulties getting existing cloud computing oriented simulators to simulate automatic scaling, and the success in verifying the load testing experiment with queue-theoretic models, we decided to implement a simulator specifically for automatic scaling scenarios using a QN model at the core instead of discrete event simulation.

By basing the simulator on queueing theory, we avoid the problem of bad queueing logic implementation in discrete event simulators. Writing it correctly is prone to errors, as we have seen on the example of CloudSim, and even if done with a due understanding of queueing theory, one will run into problems with rounding errors when small time steps

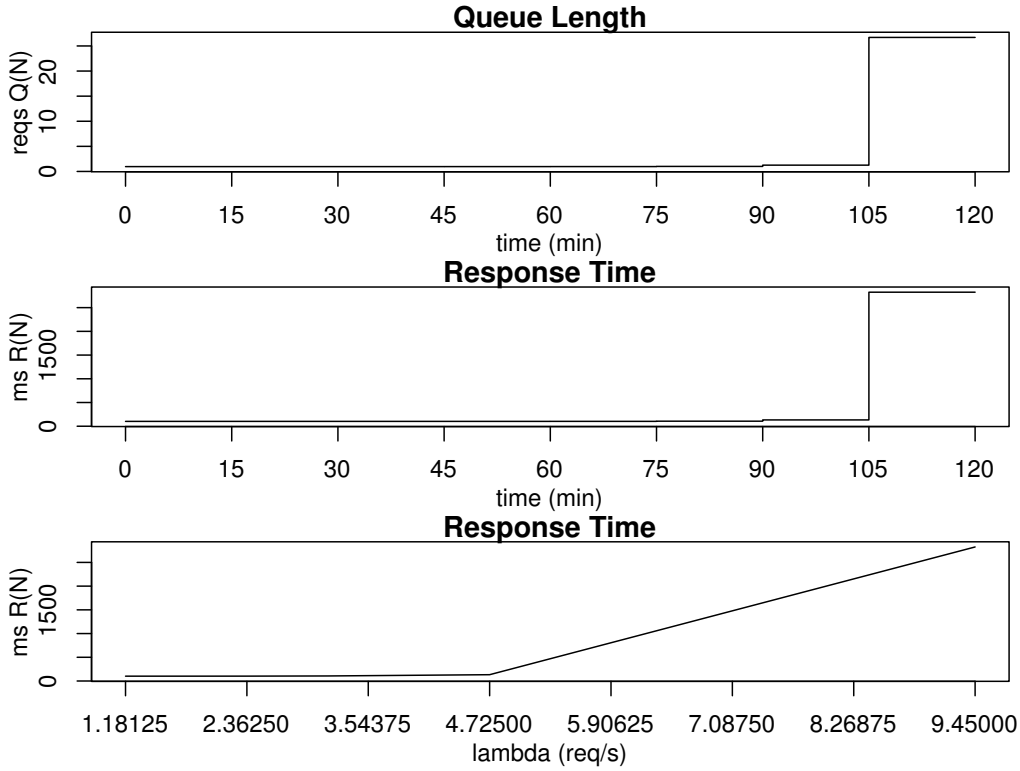


Figure 3.8: Results from PDQ in R

are used. Using a QN model as the engine will also make the simulator much smaller and faster than its discrete event counterparts.

The proposed simulator should be able to take a trace of incoming requests and compute the observed latencies based on the supplied system model. Due to the choice of the QN formalism for the model, we are theoretically limited to modeling the system in steady state. However, in practice, the QN models are commonly used to describe real systems by taking e.g. the average input request rate in a peak hour and declaring that the system was statistically steady during that hour. We argue that in real cloud systems, the input rate will change gradually, not abruptly, so that if we discretize the input time series in small enough steps, we can declare the system as statistically steady in each of those steps, which will allow us to create a QN model for every step of our simulation.

To simulate automatic scaling, our initial choice of time granularity will be 15 minutes, for multiple reasons. It is a useful constant for drawing graphs of daily load curves, which are one of the leading causes for the implementation of autoscaling. (The second cause being sudden spikes in traffic.) Our previous work [3] is concerned with prediction of daily load curves and also uses a granularity of 15 minutes, grounded in the maximum allowed lags of autoregressive time series forecasting algorithms in R. Due to most cloud computing services being billed in hourly cycles, granularity equal to a fraction of an hour makes sense in modeling of the cloud. With instance start-up times, including perhaps some DevOps

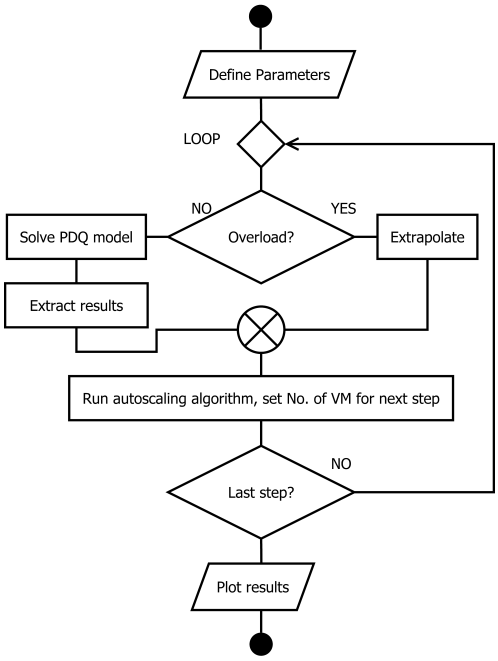


Figure 3.9: Flowchart of the autoscaling simulator

setup steps after the actual instantiation, being close to 10 minutes, and load average measurement intervals in automatic scalers defaulting to 5 minutes, this constant gives the model a realistic lag when adding new instances. Nothing is hindering users from choosing a smaller interval if they have corresponding input data.

The model is recomputed each time interval, and its output latency is given as input to an user-defined autoscaling algorithm, which decides on parameters for the next time interval, mainly the number of machines serving requests. This is equivalent to the MAPE-K (Measure, Analyze, Plan, Execute using Knowledge) loop of self-adaptive systems referenced in other articles.

A flowchart of the actual algorithm is shown in Figure 3.9. First, the model variables are initialized. The system then enters the main loop. Because the PDQ QN analyzer cannot model overload situations of open queueing networks (a system where the queue length approaches infinity is not steady), the situation when the incoming load is higher than the system capacity $\lambda > \mu$ is detected and the latency extrapolated from the last simulation step using the expression

$$Q(t) = Q(t-1) + \Delta T * (\lambda - \mu)$$

where $Q(t)$ and $Q(t-1)$ are queue lengths at current and previous time steps, respectively, ΔT is time step size, λ is incoming request rate and μ maximum service rate. This formula gives the queue length at the end of the time interval when the system was overloaded. We get the response time as

$$R = (Q+1)D$$

where $(Q + 1)$ is the queue length plus 1 for the request currently in service, under the memoryless property of exponential distribution), and D the service demand. This is a pessimistic estimate as it assigns the value at the end of an interval of overload for its whole length.

The values from the model are then extracted and fed into the autoscaling algorithm, which produces machine counts for the next iteration. When the defined simulation time is reached, charts with the results are plotted and output values printed to the terminal in textual form.

The QN solving library used is PDQ, mainly because its ease of integration into other applications and scripts. With other solvers, it would be necessary to generate model specifications in XML (in the case of JMT) or another language, execute the external application, and parse the results. PDQ can be called from within the language. The solution algorithm used is known as the Canonical method for open queueing networks. It is not exactly specified in [42], but probably is the one described in [12, page 363], based on the BCMP (Baskett, Chandy, Muntz, Palacios) and Jackson's theorems. While the library also has the exact and approximate MVA methods for closed QN systems, an open system is a better choice for cloud simulations driven by a request intensity trace. Otherwise, we would need a trace of the number of users accessing the system in time (Like the input parameters of CloudAnalyst, whose load generator creates an open flow anyway.).

The simulator script itself is written in R, as it is our platform of choice for time series analysis. It has a range of functions for data import and preprocessing, a vast library of statistical analysis tools and excellent graphing and reporting capabilities. Using the PDQ library in a cycle was inspired by a blog article by Gunther[1]. As seen there, while the first version of our script simulates a single autoscaled tier, it is trivial to extend it to simulate a multi-tier system (in fact, it has already been done, see the GitHub page [2]). Several input streams of different service demands can be created, as long as there is input data for these streams. However, with the tendency of cloud-based applications to separate different functions into different instances, we do not think the multi-stream feature will be necessary.

The input request intensity traces are expected as R time series objects. These are simple vectors of equidistant observations (they may contain multiple vectors), which can also have metadata like starting index, time units, and sampling frequency. Another input parameter is the service demand of one request (at each tier and request class). The simulator then creates similar time series traces of utilization, the number of VMs in service, and the request latency computed from the QN model.

All historical values of the traces are available to the autoscaling decision algorithm, which is written as a function block in the script. Therefore. It is possible to implement an autoscaler of Google type, operating by setting thresholds on latency, or of Amazon type, which measures average utilization in time intervals of specified length and compares its thresholds against those. The queue length trace is also available enabling the simulation

[1] `http://www.r-bloggers.com/applying-pdq-in-r-to-load-testing`

[2] `https://github.com/vondrt4/cloud-sim`

of PaaS autoscalers, which react on the queue length at the load balancer (reportedly implemented in OpenShift).

It is not possible to implement the RightScale voting protocol, which computes thresholds on each individual node and adds or removes nodes based on majority vote, as all nodes in the model have the same utilization. It cannot model effects of different load-balancing strategies because it does not work with individual requests, but averages under steady state.

What it can do is to run the simulation from an arbitrary point in the input time series, which leaves historical data to run time series forecasts. The forecast values can be used as input for autoscaler rules, making the simulator suitable for evaluation of proactive autoscaler designs. Most existing forecast algorithms are already implemented in R, rendering the task even simpler.

Chapter 4

Main Results

4.1 Feasibility of Interactive Workload Prediction

4.1.1 Loading of data

The evaluation of the forecasting methods was done on six time series from servers running different kinds of load. The data was first extracted from RRDTool and pushed into MySQL by a bash script, which was being run every day to get data at the desired resolution. The RRD format automatically aggregates data points using maximum, minimum and average, after they overflow the configured age boundaries. Those were (in files created by Collectd) 10 hours in 30 second intervals, 24 h in 60 s, 8 days in 8 minutes, 1 month in 37 min, and 1 year in 7.3 hours.

The chosen initial resolution for experiments was 15 minutes, as the aim is to forecast a) for IaaS clouds, where instance start up takes about 5 minutes, plus user initialization, and accounting is done in hours, and b) for batch jobs, where the user will probably give task durations in hours or their fractions. Later, it will be evident that this resolution is appropriate for forecasts with the horizon of days, which was the goal of the selection.

The data was then loaded into R (using manual [75]). There was a total of 8159 observations or 2.8 months of data. Time series objects (ts) were created. Their drawback is that observations need to be strictly periodic and the x axis is indexed only by numbers. Any missing values have been interpolated (there was no larger consecutive missing interval). For uneven observation intervals, the "zoo" library may be used, which indexes observations with time stamps [76]. It was not used here, so for clarification: The measurement interval starts with time stamp 1128, which was November 28, and then the count increases every day by 1 irrespective of the calendar as the seasonal frequency was set to 1 day. Therefore, the interval contains Christmas at about 1/3, and it ends on Thursday.

4.1.2 Time series diagnostics

The servers included in the experiments have code names oe, bender, lm, real, wn, gaff. In the next paragraph follow their designations and the result of examinations of the time plots of their CPU load time series. These series were also filtered by simple moving average (SMA) with window set to 1 day to obtain deseasonalized trend. The time plots of the series along with best forecasts from both methods are attached in Subsection 4.1.7.

oe is a large web shop. It has a clear and predictable daily curve with one weekday higher and weekend and holidays lower (incl. Christmas). Trend is stationary (except Christmas).

bender is shared PHP webhosting. It has a visible daily curve with occasional spikes. First month shows a decreasing trend, and then it stabilizes.

ln is a discount server. The low user traffic creates a noisy background load that is dominated by spikes of periodic updates. Trend alternates irregularly between two levels; the duration is on the scale of weeks.

real is a map overlay service, not much used but CPU intensive (as one map display operation fetches many objects in separate requests). The time plot is a collection of spikes, more frequent during day than night. There are 2 stationary levels, where the first month the load was higher, and then the site was optimized so it went lower.

wn is PHP hosting of web shops. It has low traffic with a visible daily curve. There is a slow linear additive trend after the first month.

gaff is a web shop aggregator and search engine. Its daily curve is inverted with users creating background load in the day and a period of high activity due to batch imports during the night. Trend is stationary.

As suggested in the tutorial by Coghlan [77], which also covers installation of R and packages, as well as Holt-Winters and ARIMA models, the time series were run through seasonal decomposition. For oe, bender and wn, the daily curve was as expected; with gaff, the nightly spike also showed nicely. lm and real surprisingly also show daily seasonality as the spikes are apparently due to periodic jobs. Decomposition of the first month of oe is in Figure 4.1. We can clearly see the repeated daily curve and a change in trend during Christmas.

Another tool to diagnose time series is the seasonal subseries plot. When applied to the test data, only oe shows clean seasonal behavior. In the bender series, noise may be more dominant than seasonality. The lm series seasonal subseries is also not clearly visible. real clearly shows that traffic on certain hours is higher. For wn, the upward trend is visible in each hourly subseries. gaff shows that the duration of the batch jobs is not always the same so there are large spikes in the morning hours, mainly at the start of the measurement interval. This plot is in Figure 4.2. It contains 96 subseries because of the 15 min frequency, index 0 is midnight.

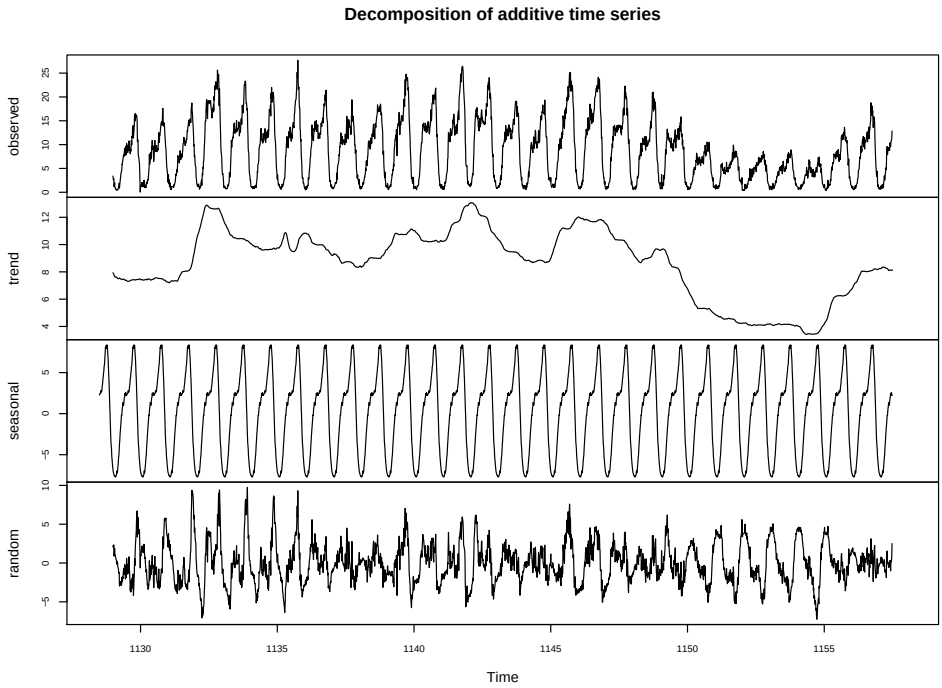


Figure 4.1: oe series decomposition, from top to bottom: overall time plot, trend, seasonal and random compoment

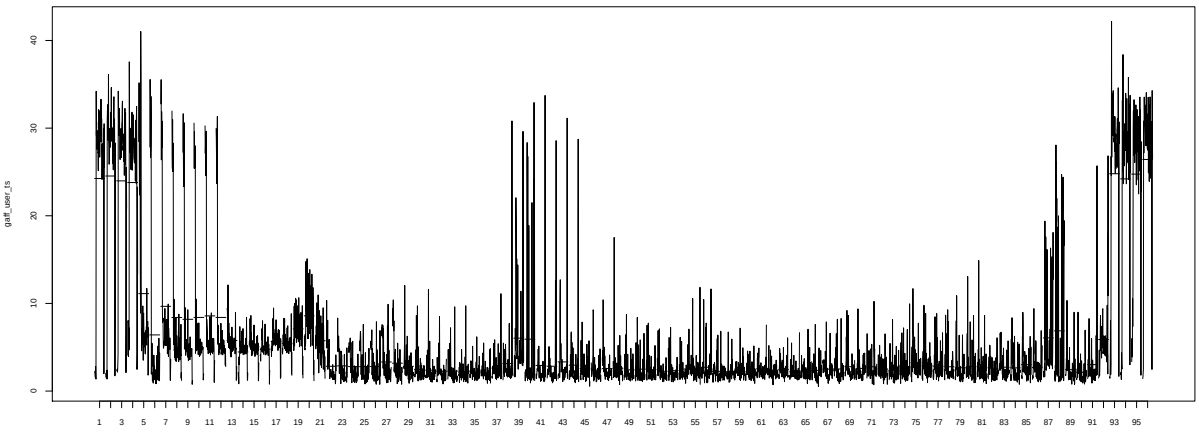


Figure 4.2: gaff series seasonal subseries plot

4.1.3 Holt-Winters model fitting and evaluation

A modified script from Hyndman and Athanasopoulos [68, chapter 8/9] was used for model fitting and validation. The algorithm first shortens the time series by 3 days at the end and fits a model on it. Then forecasts are created for 6, 24, and 96 hour horizons and compared with the withheld validation data. The result is a table of standard model efficiency measures for each series and interval ("in" meaning in-sample). One more measure was defined in accordance with the goal specified at the beginning of this section – how many validation data points missed the computed 80% prediction intervals in the 3-day forecast (that is 288 points in total).

As to the forecast error measures, the following ones are used:

ME The Mean Error is a measure of error in absolute scale; it is signed, so it can be used to see a bias in forecasts, but cannot be used for comparison of time series with different scale.

RMSE The Root Mean Squared Error measures squared error and is thus more sensitive to outliers. It is best used when the scale of errors is significant. The square root operation returns the dimension to that of the original data.

MAE Mean Absolute Error is similar to ME, but ignores the direction of the error by using absolute values.

MPE Mean Percentage Error removes the influence of scale from ME by dividing error by the value,

MAPE Mean Absolute Percentage Error does the same to MPE. It is probably the best measure for human evaluation.

MASE The Mean Absolute Scaled Error is different from the others in that it does not compare the error to the original data, but to the error of the naïve "copy the previous value" forecast method.

For one-step-ahead forecasts, MASE values below one indicate that the evaluated method is better. For larger horizons, this is not true, as the naïve method has more information than the one under evaluation (i.e., always the previous data point). Normally, ME, RMSE, and MAE have the dimension of the original data, MPE and MAPE are in percent and MASE is dimensionless. Here, all values are dimensionless as the input data is a time series of CPU load percentages.

The result can be seen in Table 4.1. For lm, two result sets are included. The first is from a triple exponential smoothing model, but as there was a spike at the end of the fitting data, the function predicted an upward trend while the data was in fact stationary. Simple exponential smoothing was then tried, which gave lower error measures and fewer points outside confidence intervals.

A similar problem existed with real. The spikes predicted by the seasonal model missed the actual traffic spikes most of the time. It seems that the series is not seasonal after all,

Table 4.1: Evaluation of the Holt-Winters model on out-of-sample data

	ME	RMSE	MAE	MPE	MAPE	MASE	miss		ME	RMSE	MAE	MPE	MAPE	MASE	miss
oe in	0.003	1.109	0.798	2.776	17.91	1.036		**rea1 in**	-0.03	4.836	3.029	-18.2	38.47	0.398	
oe 6	0.691	1.11	0.829	6.111	7.623	1.076		**rea1 6**	-1.54	7.206	4.878	-68.3	86.06	0.641	
oe 24	0.5	2.461	1.985	-30.4	62.34	2.575		**rea1 24**	-0.28	6.843	4.77	-50.2	71.75	0.627	
oe 96	1.843	4.238	3.223	-26.1	75.49	4.181	2	**rea1 96**	-0.31	7.004	4.916	-56.3	77.77	0.646	84
bend in	-0.06	1.699	1.176	-7.38	23.45	1.11		**rea2 in**	-0.11	7.515	5.973	-68.4	95.76	0.785	
bend 6	0.015	1.28	1.068	-2.11	14.47	1.009		**rea2 6**	-1.78	6.866	5.485	-105	122.1	0.721	
bend 24	-0.36	1.436	1.2	-17.9	27.39	1.133		**rea2 24**	-0.28	8.304	6.619	-88.9	115.6	0.87	
bend 96	-1.33	2.385	1.934	-35.7	41.42	1.826	2	**rea2 96**	-0.3	8.387	6.713	-95.1	122	0.883	44
lm1 in	-0.35	5.408	3.832	-10.3	31.63	0.801		**wn in**	-0.01	2.469	1.6	-15.2	43.31	1.047	
lm1 6	3.408	4.839	3.713	18.09	20.46	0.777		**wn 6**	-0.35	1.88	1.553	-11.4	24.44	1.016	
lm1 24	-12.9	17.78	14.81	-119	129.4	3.099		**wn 24**	-1.42	3.617	2.98	-74.8	87.18	1.95	
lm1 96	-27.2	32.23	27.86	-248	251.4	5.83	97	**wn 96**	-1.29	5.151	3.995	-86.4	102.8	2.614	0
lm2 in	0.002	5.638	3.856	-13.5	31.52	0.806		**gaff in**	-0.01	3.562	2.039	-8.9	57.79	1.158	
lm2 6	0.639	5.667	4.625	-6.41	28.14	0.967		**gaff 6**	0.191	7.099	6.449	63.97	465.5	3.663	
lm2 24	-1.04	6.939	5.104	-24.7	40.55	1.068		**gaff 24**	-0.01	6.835	4.308	-8.97	189.3	2.447	
lm2 96	-1.04	7.666	5.624	-29.4	45.83	1.176	14	**gaff 96**	0.622	5.927	4.002	5.364	157.7	2.274	4

but rather cyclic. The cause for the spikes is random arrivals of requests, as per queuing theory. Cyclicity is discussed in Hyndman [78]. The important outcome is that exponential smoothing models cannot capture it, while autoregressive models can.

The second model for real in the table is double exponential smoothing, which, interestingly, shows higher error measures, but lower number of missed observations. The cause is that the confidence intervals are computed based on the variance of in-sample errors. Therefore, the closer the error magnitude is between in-sample and out-of sample measurement, the more accurate the model is in the "misses" measure.

Automatic model fitting also failed for gaff. The transition from the nightly spike to daily traffic caused the predicted values to be below zero. A manual adjustment of Alpha parameter was necessary. Computed $\alpha = 0.22$, set $\alpha = 0.69$. The problem probably is that the algorithm optimizes in-sample squared error (MSE) and thus it preferred a slower reaction, which mostly missed the spike. The computed trend from this mean was therefore strongly negative. A quicker reaction to the change in mean improved the model, but even then, series with abrupt changes in mean are not good for the Holt-Winters model.

From Table 4.1, we can see that with the Holt-Winters method, some series are predicted well even for the 3 day interval (bender, lm method 2), for some, the forecast is reasonably accurate for the first 6 hour interval and then deteriorates (oe, lm method 1, wn), for others it is inaccurate (real, gaff).

In addition, when the error measures for in-sample data are worse than for out-of-sample, it is a sign of overtraining - the validation data set was closer to "average" than the training data. This is because we were training on a long period including Christmas and verifying on a normal week. Perhaps shortening the training window would be appropriate.

The tutorial [77] suggests using autocorrelation plot on the residuals of the Holt-Winters model. A significant autocorrelation of the residuals means that they have a structure to them and do not follow the character of white noise. All the models showed significant

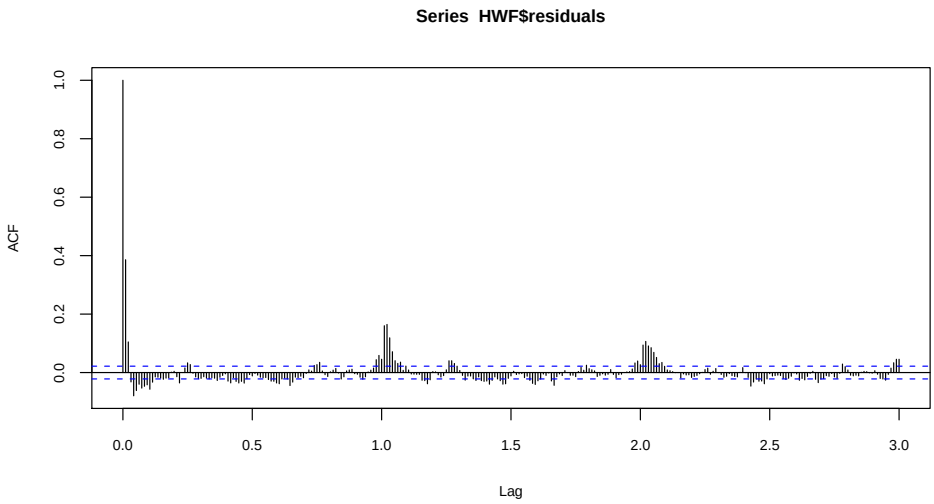


Figure 4.3: Autocorrelogram of residuals of the H-W model on bender

autocorrelation of residuals at both low lags and lags near the period. The Ljung-Box test is a more rigorous proof of randomness of a time series as its null hypothesis is that a group of autocorrelations up to a certain lag is non-significant. It can thus ignore a random spike in the ACF. All the models failed the test in the first few lags.

Having seen autocorrelation plots such as in Figure 4.3, it was decided to move to better, autoregressive, models.

4.1.4 ARIMA model selection

4.1.4.1 Differencing order

The prerequisite for ARIMA is that the time series is stationary. Manually, stationarity can be detected from the time plot. A stationary time series has constant level and variance, and may not exhibit trend or seasonality. The two last effects should be removed for identification of model order, but are covered by ARIMA models with non-zero differencing order and SARIMA (Seasonal ARIMA), respectively. For series with non-linear trend or multiplicative seasonality, the Box-Cox transform should be used, but that was not the case with the series studied here. Additionally, a non-stationary series will have ACF or PACF plots that do not decay to zero.

The statistical approach to identification of differencing order is through unit root tests (see Nielsen [79]). The root referred to here is the root of the polynomial function of the autoregressive model. If it is near one, any shocks to the function will permanently change the level and thus the resulting series will not be stationary. The standard test for this is Augmented Dickey-Fuller (ADF), which has the null hypothesis of unit root. A reversed test is Kwiatkowski-Phillips-Schmidt-Shin (KPSS), where the null hypothesis

is stationarity. There is also a class of seasonal unit root tests that can help specify the differencing order for SARIMA, these are Canova-Hansen (CH) and Osborn-Chui-Smith-Birchenhall (OCSB).

Table 4.2: Order of differencing based on unit root tests

	oe	**bender**	**lm**	**lm4**	**real**	**real4**	**wn**	**gaff**
ADF	0	0	0	0	0	0	0	0
KPSS	0	1	1	1	1	1	1	0
OCSB	0	0	0	0	0	0	0	0
CH	0	0	0	1	0	1	0	0

In R, there exist functions ndiffs() and nsdiffs(), which automatically search for the differencing and seasonal differencing order, respectively, by repeatedly using these tests and applying differences until the tests pass (for KPSS and CH), or stop failing (for ADF and OCSB). The default confidence level is 5%. The recommended amount of differencing of the experimental time series obtained from the tests is in Table 4.2. Columns lm4 and real4 will be explained later.

It is evident that the ADF and KPSS tests did not agree with each other with the exception of oe and gaff. According to [79], ADF should be considered primary and KPSS confirmatory. The same is said by Stigler in discussion [80], adding that unit root tests have lower sensitivity than KPSS. In the same discussion, Frain says KPSS may be more relevant as a test concretely for stationarity (there may be non-stationary series without a unit root), if we do not assume a unit root based on underlying theory of the time series. It was also used by Hyndman in the auto.arima() function for iterative model identification.

According to manual heuristic approaches, such as presented by Nau [81], an order of seasonal differencing should always be used if there is a visible seasonal pattern. It also suggests applying a first difference if the ACF does not decay to zero. An example of the impact of first and seasonal differencing on stationarity and thus legibility of an ACF plot is in Figure 4.4.

The ACF and PACF functions on the test data were looked at with and without differencing with the result that differencing rapidly increases the decay of the ACF function on all series except real.

Moreover, from the ACF of lm and real, it seems there is a strong periodicity of 4 hours. These two series will be also tested with models of this seasonal frequency and will be denoted as lm4 and real4, as in Table 4.2.

For the purpose of order identification, seasonal and then first differences have been taken. It was decided to test if the models fitted with this order of differencing, following the heuristic approach, are better or worse than those with differencing order identified by statistical tests.

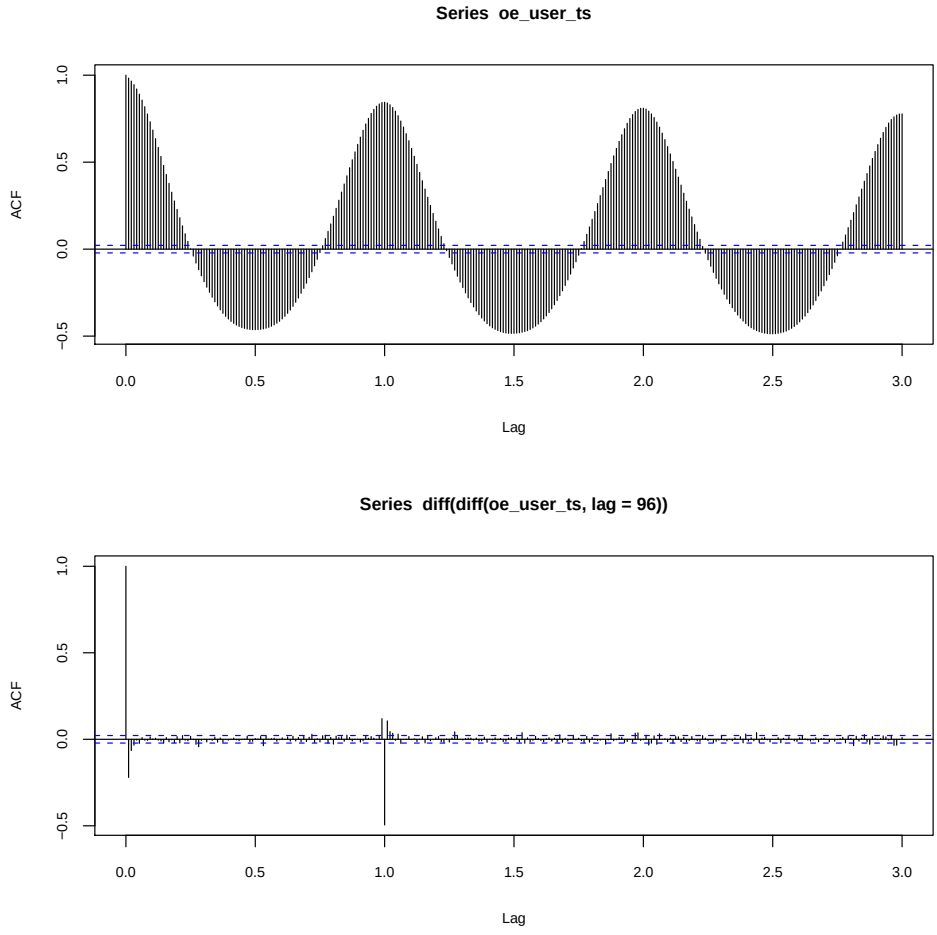


Figure 4.4: ACF of oe without and with differencing

4.1.4.2 Order identification

Identification of model order was done using heuristic techniques from [68, 70, 81, 82]. After seasonal and first differencing is applied in the necessary amount to make the time series look stationary to the naked eye, so that its autocorrelograms converge to zero, the ACF and PACF functions are looked at. The number of the last lag from the beginning where PACF is significant specifies the maximum reasonable order of the AR term, similarly the last significant lag on ACF specifies the MA order. The order of the seasonal autoregressive and moving average terms is obtained likewise, but looking at lags that are multiplies of the seasonal period.

The observed last significant lags and resulting maximum model orders are summed in Table 4.3. Model parameters are denoted as ARIMA(p, d, q)(P, D, Q), where p is the order of the AR term, d is the amount of differencing and q is the order of the MA term. The second parenthesis specifies the seasonal model orders.

Table 4.3: Last significant lags and model orders

	PACF	ACF	seas. PACF	seas. ACF	est. max. model params
oe	5	3	11	1	ARIMA(5,1,3)(11,1,1)
bender	17	4	9	1	ARIMA(17,1,4)(9,1,1)
lm	15	16	8	1	ARIMA(15,1,16)(8,1,1)
lm4	9	2	11	∞	ARIMA(9,1,2)(11,1,0)
real	1	2	11	1	ARIMA(1,0,2)(11,1,1)
real4	13	2	11	1	ARIMA(13,1,2)(11,1,1)
wn	39	3	10	1	ARIMA(39,1,3)(10,1,1)
gaff	18	2	6	1	ARIMA(18,1,2)(6,1,1)

Looking at the two variants of lm, the expectation is that the first will perform better, as the non-seasonal part covers the second period of 4 hours. This is not true for real vs. real4.

4.1.4.3 Model estimation

When trying to fit models with high seasonal order, a limitation of the ARIMA implementation in R was found. The maximal supported lag is 350, which with a period of 96 (24 hours * 4 observation per hour) means that the seasonal lag is limited to 3.

Furthermore, the memory requirements of seasonal ARIMA seem to be exponential with the number of data points. A machine with 1 GB of RAM could not handle the 2.8 months of data with lag 288. This constraint is not documented. The experiment had to move to a machine with 32 GB RAM, where computing a model with seasonal order 3 took 7.6 GB RAM, more on subsequent runs as R is a garbage collected language.

For the course of this experiment, the order of the seasonal components will be limited to three, as it should be sufficient when forecasting for a horizon of about a day. The alternatives, which will be examined in further experiments, are to reduce the resolution to 1 hour, which will enable lags up to 12 days.

A model of this sort was fitted on oe, and it did not lead to a better expression of the weekly curve (at least not by visual inspection). With this resolution, it will be however possible to use a seasonal period of one week, which should be able to capture the day-to-day fluctuations. Similarly, we would reduce resolution is we were trying to capture monthly or yearly seasonality.

Another approach, suggested by Hyndman [83], is to model the seasonality using a Fourier series and to use non-seasonal ARIMA on the residuals of that model. This should enable fitting on arbitrarily long seasonal data. This may lead to overfitting, though, as the character of the time series is subject to change over longer time periods.

For the actual parameter estimation, the Arima() function with the model order as parameter can be used. There is however a way to automate a part of the identification-estimation-validation cycle and that is the auto.arima() function. This function repeatedly fits models with different parameters and then returns the one that has minimal Akaike Information Criterion (AIC). This criterion prefers models with lower likelihood function and contains a penalization for the number of degrees of freedom of the model; therefore, it should select the model that best fits the data, but not variations of the same model with superfluous parameters.

The auto.arima() function has two modes depending on the "stepwise" parameter (see help(auto.arima) in R). With this set to TRUE, it does a greedy local search, which selects the best model from previous step and examines its neighborhood in the state space given by adding or subtracting one to each parameter. It continues, until no model in the neighborhood has lower AIC.

The second mode searches from ARIMA(0,0,0)(0,0,0) upwards and based on the description, it should search until the ceiling set for each parameter. The actual behavior however seems to be that is stops when the last iteration examined did not bring any gain. Both search modes are thus prone to getting stuck in a local minimum.

To better specify the models, the auto.arima() function was used on each time series with three sets of parameters. In the first run, it was started from zero with stepwise=FALSE and with ceilings set to the parameters estimated in Table 4.3. In the second run, stepwise was set to TRUE and the ceilings were left at the pre-estimated parameters plus one to account for differencing; the starting values were set to be the same as the ceilings, as, theoretically, the parameters in Table 4.3 should be the maximal meaningful numbers, but a model with lower orders might be better. This was tested in the third run, where the starting values remained and the ceilings were effectively removed.

The same procedure was then repeated with the differencing orders computed by OCSB and KPSS. As it is difficult to identify model parameters by naked eye without differencing, the same initial parameters have been used. Please note that the AIC values of models with unequal differencing order are not comparable, while goodness-of-fit test results and prediction errors are.

4.1.5 Model validation

As already discussed in Subsection 3.1.2 "Box-Jenkins models", the validation entails manual examination of the autocorrelation plot of residuals and use of the Ljung-Box goodness-of-fit (GOF) test. Table 4.4 contains the models that resulted from the three runs of auto.arima() as described, along with their AIC values, the lag of the first significant autocorrelation and the lag after which the Ljung-Box test failed. Left side is for models with differencing order set to one, right side has differencing set by unit root tests.

The outcome from Table 4.4 is, that is cannot be conclusively said whether it is better to always use seasonal differencing or not. Of the six time series, three have the best fitting model in the left half of the table and three in the right half. However, it seems that in the cases where the non-differenced models were better, the gain in the goodness-of-fit

Table 4.4: Parameters of the estimated ARIMA models and their validation measures

	model	AIC	sig. ACF	fail. GOF		model	AIC	sig. ACF	fail. GOF
oe	ARIMA(0,1,2)(1,1,2)	23016.97	12	14	**oe**	ARIMA(4,0,3)(2,0,2)	23231.26	22	23
	ARIMA(6,1,1)(1,1,2)	23109.12	12	15		ARIMA(5,0,4)(3,0,2)	23259.8	21	27
	ARIMA(5,1,3)(2,1,3)	23066.43	16	20		*ARIMA(5,0,3)(3,0,3)*	*23220.86*	*21*	*27*
bend	ARIMA(1,1,1)(2,1,1)	28082.19	4	6	**bend**	ARIMA(1,1,1)(3,0,2)	27989.07	22	6
	ARIMA(17,1,5)(3,1,2)	27580.45	52	129		ARIMA(17,1,4)(3,0,2)	27812.25	26	60
	ARIMA(17,1,3)(3,1,3)	*27504.15*	*58*	*172*		ARIMA(14,1,1)(3,0,3)	27801.06	17	58
lm	ARIMA(1,1,1)(2,1,1)	47569.93	5	5	**lm**	ARIMA(1,1,3)(1,0,2)	48517.21	5	4
	ARIMA(16,1,17)(2,1,2)	47210.57	94	144		ARIMA(15,1,17)(3,0,1)	48155.89	43	144
	ARIMA(17,1,17)(2,1,3)	*47195.88*	*98*	*500+*		ARIMA(15,1,18)(3,0,1)	48152.6	70	144
lm4	ARIMA(2,1,1)(1,1,1)	49151.84	5	5	**lm4**	ARIMA(1,1,3)(0,0,2)	50789.93	4	4
	ARIMA(10,1,3)(12,1,1)	48398.45	11	14		ARIMA(10,1,2)(12,0,2)	48570.04	10	28
	ARIMA(11,1,2)(16,1,5)	48138.57	21	30		ARIMA(12,1,2)(15,0,4)	48342.89	21	30
real	ARIMA(1,1,1)(2,1,1)	47872.32	4	3	**real**	ARIMA(0,1,2)(3,0,0)	48873.62	4	4
	ARIMA(2,1,3)(2,1,2)	47800.56	1	1		ARIMA(2,1,3)(3,0,2)	48732.74	1	1
	ARIMA(6,1,8)(3,1,3)	46972.94	6	6		ARIMA(10,1,12)(3,0,3)	47344.03	8	9
rea4	ARIMA(2,1,1)(1,1,1)	47574.67	3	3	**rea4**	ARIMA(1,1,1)(3,0,0)	48897.42	3	3
	ARIMA(1,1,3)(1,1,2)	47612.58	2	2		*ARIMA(12,1,2)(11,0,1)*	*47438.97*	*21*	*24*
	ARIMA(4,1,5)(1,1,7)	47373.03	8	7		*ARIMA(12,1,2)(11,0,1)*	*47438.97*	*21*	*24*
wn	ARIMA(4,1,2)(2,1,2)	35599.79	9	14	**wn**	ARIMA(2,1,3) with drift	36214.64	10	9
	ARIMA(40,1,2)(2,1,2)	35608.54	55	500+		ARIMA(39,1,4)(1,0,2)	36177.56	59	95
	ARIMA(39,1,1)(2,1,3)	*35596.67*	*55*	*500+*		ARIMA(38,1,5)(1,0,3)	36146.1	64	191
gaff	ARIMA(2,1,3)(0,1,2)	21501.92	5	5	**gaff**	ARIMA(3,0,0)(1,0,1)	21847.8	5	5
	ARIMA(19,1,3)(0,1,2)	21387.26	42	52		*ARIMA(18,0,3)(1,0,2)*	*21717.96*	*43*	*88*
	ARIMA(17,1,4)(0,1,3)	21118.64	42	88		*ARIMA(18,0,3)(1,0,2)*	*21717.96*	*43*	*88*

functions was lower than the other way round. It is also interesting that in two of the three cases (oe and gaff), the difference is not only in seasonal, but also in first differencing. It may be a good idea to follow the recommendation of the KPSS test, but always use seasonal differencing, but there is not enough data to say it with certainty.

A more solid fact is that all the best models come from the third row of the table. Of the three tried here, the best algorithm for model selection is to use auto.arima() in greedy mode, starting with parameters identified from ACF and PACF, and leave it room to adjust the parameters upwards.

4.1.6 Comparison of the two model families

The last part of the experiment entailed computing forecasts based on the fitted ARIMA models and comparing them with out-of-sample data. The same validation algorithm was used as in the case of Holt-Winters models, to facilitate model comparison. The result is in Table 4.5. To conserve space, only MAPE (Mean Average Percentage Error) is shown. The four columns are for in-sample error and forecast errors in horizons 6, 24, and 96 hours. The ordering of models is the same as in Table 4.4.

Fitting of the forecasts was something of a disappointment, as all of the models with seasonal differencing (the left half of Table 4.4) that were selected as best using the GOF measures have failed to produce forecasts. The cause was likely the seasonal MA part of the

Table 4.5: Evaluation of the ARIMA models on out-of-sample data

	MAPE in	MAPE 6	MAPE 24	MAPE 96	miss		MAPE in	MAPE 6	MAPE 24	MAPE 96	miss
oe	13.43	*7.12*	75.27	94.99	8	**oe**	13.4	8.13	37.52	53.16	22
	13.39	7.13	77.74	98.23	*7*		13.22	8.71	33.96	48.88	22
					failed		*13.16*	8.85	*31.41*	*46.35*	24
bend	19.32	*14.56*	22.18	22.21	25	**bend**	*18.28*	15.34	45.21	41.32	*13*
	18.51	15.51	*20.58*	*21.3*	42		18.79	21.42	36.53	34.38	87
					failed						failed
lm	24.93	19.14	*19.7*	*22.34*	17	**lm**	25.49	17.23	19.8	23.22	19
	23.94	14.79	20.1	25.15	20		23.98	15.74	21.01	26.91	21
					failed		23.97	15.98	21.11	27.02	21
lm4	25.5	13.22	23.36	28.93	8	**lm4**	28.77	26.94	44.3	51.17	*7*
	24.73	*11.66*	22.19	30.21	24		24.61	12.93	21.81	29.02	21
	24.16	15.57	20.29	23.45	19		24.47	12.51	19.88	25.73	21
real	36.26	85.66	73.58	80.2	85	**real**	38.18	72.49	62.91	64.49	59
	37.13	88.91	76.86	83.95	94		38.18	87.33	74.95	81.07	81
					failed		37.56	69.18	52.74	58.08	*39*
rea4	37.67	58.08	48.3	54.23	59	**rea4**	40.83	53.41	47.86	70.73	43
	37.99	53.44	43.22	*45.75*	40		*36.1*	*50.4*	*41.59*	46.06	49
	37.03	55.33	43.54	46.34	61		*36.1*	*50.4*	*41.59*	46.06	49
wn					failed	**wn**	42.03	*24.3*	78.8	82.33	102
	37.68	36.26	*51.12*	*50.25*	*59*		38.92	24.36	64.94	71.68	79
					failed		38.96	26.33	64.16	70.09	78
gaff	*37.62*	*160.67*	128.23	112.77	61	**gaff**	37.65	170.08	136.91	124.57	*59*
	38.19	165.29	*125.89*	*109.42*	60		38.26	165.48	133.85	119.44	*59*
	38.56	187.57	129.88	110.55	61		38.26	165.48	133.85	119.44	*59*

model that was one or two orders higher that the originally identified ceiling. That resulted in an overspecified model where the MA polynomial was not invertible. Invertibility is a prerequisite for the computation of variances of the parameters [84], which in turn are needed to compute confidence intervals for a prediction. Hence, these models were fitted and had a likelihood function and in-sample errors, but could not be used for forecasts with confidence bounds.

When fitting ARIMA models in R, one needs to carefully observe the output for warnings such as:

```
In sqrt(z[[2]] * object$sigma2) : NaNs produced
for least-squares fitting, or for maximum likelihood:
Error in optim(init[mask], armafn, method = optim.method, hessian = TRUE,  :
non-finite finite-difference value [1]
In log(s2) : NaNs produced
```

because then the prediction will produce wrong results or fail:

```
MA part of model is not invertible
```

Therefore, if using auto.arima() beyond the ceiling identified from ACF and PACF, there is a high risk of the model failing and thus it may not be a good idea for automatic forecasts. If that happens, lowering the order or the seasonal MA or MA part should help.

As to the selection of the best model for forecasts, the selection based on out-of sample forecast errors (mainly looking at the 24 and 96-hour horizons) corresponds to the one based on goodness-of-fit criteria. In the case where the model fails to produce forecasts, the next-best one based on GOF can be selected. The second row (ceilings from ACF and PACF adjusted downward by auto.arima()) produced the best result, except on oe and lm, where, however, the difference is seems to be small.

As whether to always use seasonal differencing, the experiment is inconclusive. In the case of oe, there was a significant gain in accuracy by not using it, in the case of wn and bender, the opposite is true.

Looking at the "misses" criterion, one could say that Holt-Winters is better. However, that outcome might be skewed. The criterion counts the number of data points that missed the 80% confidence bounds in the 3-day forecast. That time period contains a total of 288 points, 20% of that is 57.6, and that is the count of data points that are by definition allowed to miss the bounds.

Therefore, the result of this comparison is that the confidence bounds on ARIMA are more accurate, or at least tighter than on Holt-Winters. If this method is to be used as proposed by this article, the confidence level used has to be adjusted upwards to 95 or 99%, depending on the overload sensitivity of the computer infrastructure.

Comparing the two model families using the MAPE error measure, the outcome is that ARIMA did produce better forecasts than Holt-Winters, except for the 6-hour forecasts on oe and bender, and also that simple exponential smoothing outperformed both seasonal methods on 24 and 96-hour forecasts on lm.

As it is expected that the cloud will contain mostly load-balanced web servers as the variable component, we think that these methods are viable for further research in the optimization of cloud computing.

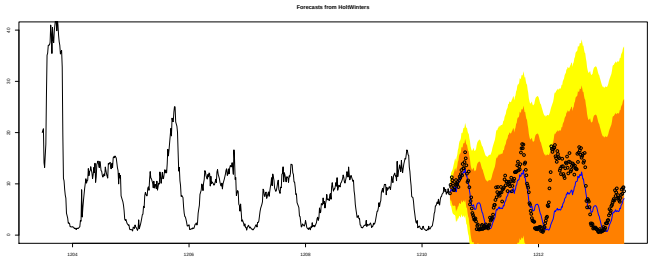
4.1.7 Forecast plots

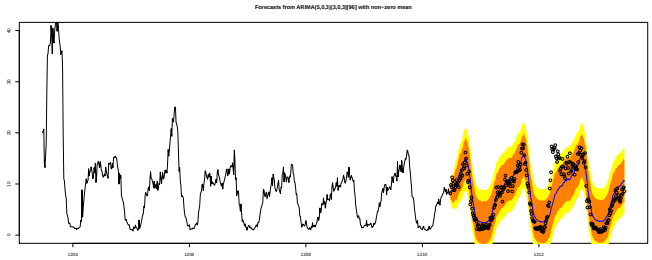
The next page contains the forecasts of each examined time series from the best model of exponential smoothing and ARIMA methods.

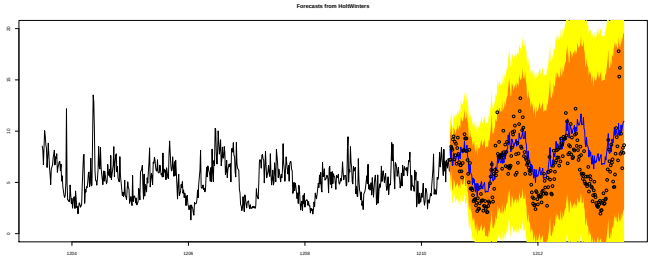
The exponential smoothing on in the left half of the page, ARIMA on the right. The series are, from top to bottom: oe, bender, lm, real, wn, gaff.

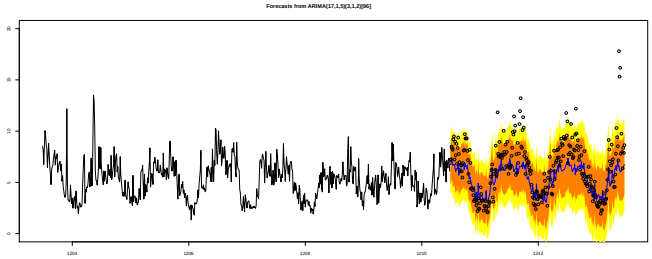
The graphs contain the last week of the time series to present their character. The blue line then represents the point forecasts; the orange area is the 80% confidence band and the yellow area the 95% confidence band. Overlaid as "o" symbols are the actual data points, which were recorded during the forecast horizon.

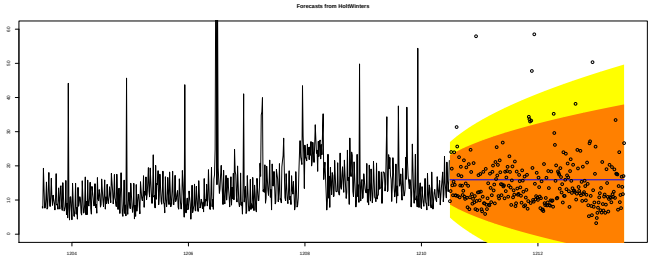
It is not important to read the axes of the graphs, the scale is 2 days per tick on the x axis and in percent of an unspecified CPU on y. The character of the time plot and the response of the forecasting algorithms is important.

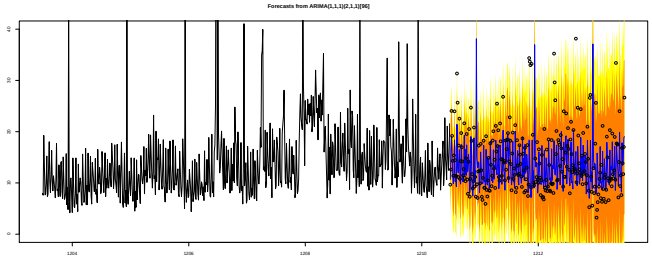

Forecasts from HoltWinters

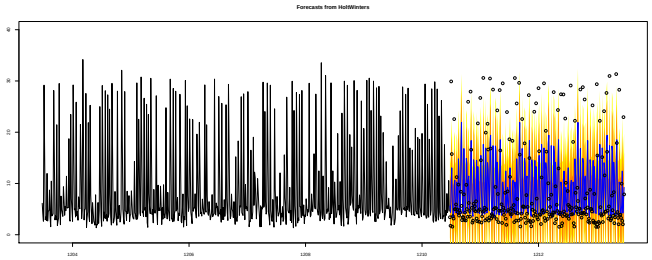

Forecasts from ARIMA(5,0,3)(3,0,3)[96] with non-zero mean

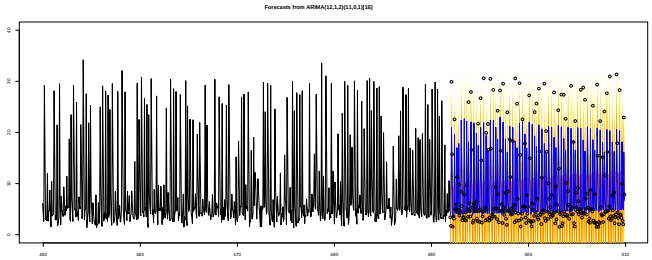

Forecasts from HoltWinters

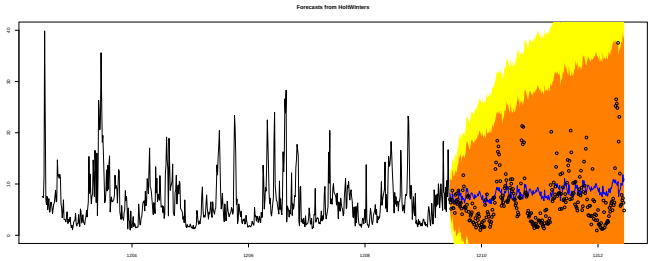

Forecasts from ARIMA(17,1,5)(3,1,2)[96]

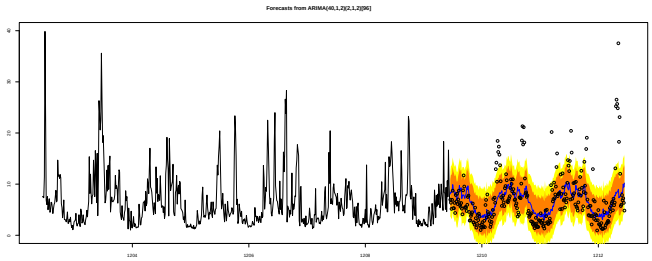

Forecasts from HoltWinters

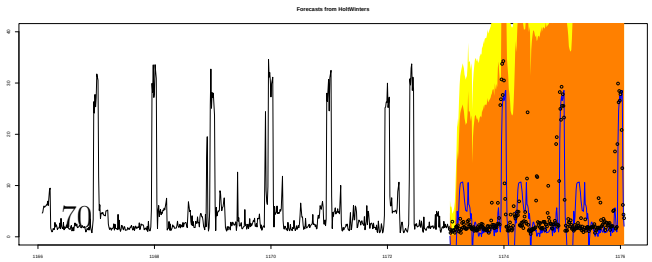

Forecasts from ARIMA(1,1,1)(2,1,1)[96]

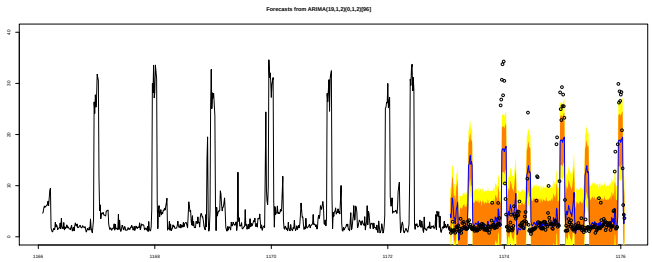

Forecasts from HoltWinters


Forecasts from ARIMA(12,1,2)(1,0,1)[16]


Forecasts from HoltWinters


Forecasts from ARIMA(40,1,2)(2,1,2)[96]


Forecasts from HoltWinters


Forecasts from ARIMA(10,1,2)(0,1,2)[96]

4.2 CloudSim Modifications for Interactive Traffic

This section presents some details about the implementation of the VM addition function and the results of the autoscaler that was implemented upon it. Figures 4.5 and 4.6 present the class diagrams of the CloudAnalyst extension and CloudSim, respectively. The classes modified during this work are highlighted.

To be concrete, DataCenterBroker contains code to add and remove VMs, DataCenterController contains the autoscaler and its latency and throughput counters, and class Extra was added to provide hourly workload levels. In the second part of the section, which focuses on improving accuracy, we added CloudletSchedulerFIFO, worked on CloudletSchedulerTimeShared and DataCenter, which contain computation of cloudlet duration, and UserBase, which is the load generator.

4.2.1 Autoscaler implementation in CloudSim

4.2.1.1 VM Addition Implementation

To be able to add Virtual Machines dynamically, it was necessary to simulate the messages received by the DataCenterController from the DataCenterBroker, which controls where the VMs are instantiated. To do so, code was added into the main loop the DataCenterController entity, triggering a function as follows:

```
boolean flag=true;
  while(Sim_system.running()){
    if(flag && GridSim.clock()>8*Constants.MILLI_SECONDS_TO_HOURS){
      flag=false;
      this.addVMs(5);
  }}
```

This code will call the function addVMs() in DataCenterBroker, when the simulation time reaches 8 hours. Triggering a function like this does not guarantee that it will wake up exactly at the specified moment, but as the frequency of events is rather high, it did not pose a problem. The entities in SimJava are long-running Java threads that react to messages. Only some classes are entities and can process simulation events. That means, *e.g.*, that a Host or Cloudlet cannot actively send a message, all initiative must be programmed in the Datacenters, User Bases, and Brokers. It also makes the code for request handling very complex and rather illegible.

The function addVM(int number) is based on the original submitVMList(), but with slight modifications. It defines the characteristics of the new VMs that will be created, and adds them to the existing list of VMs vmlist. Then, it modifies the vector of VM to Datacenter mappings, setting it to -1 for all the new machines. Then the function processMyResourceCharacteristics() is used to call CreateVMInDatacenter(0), shortcutting the decision functionality of the broker and scheduling a message to the Datacenter entity number 0. Lastly, we will create a 3-position vector for each of the new VMs, where the

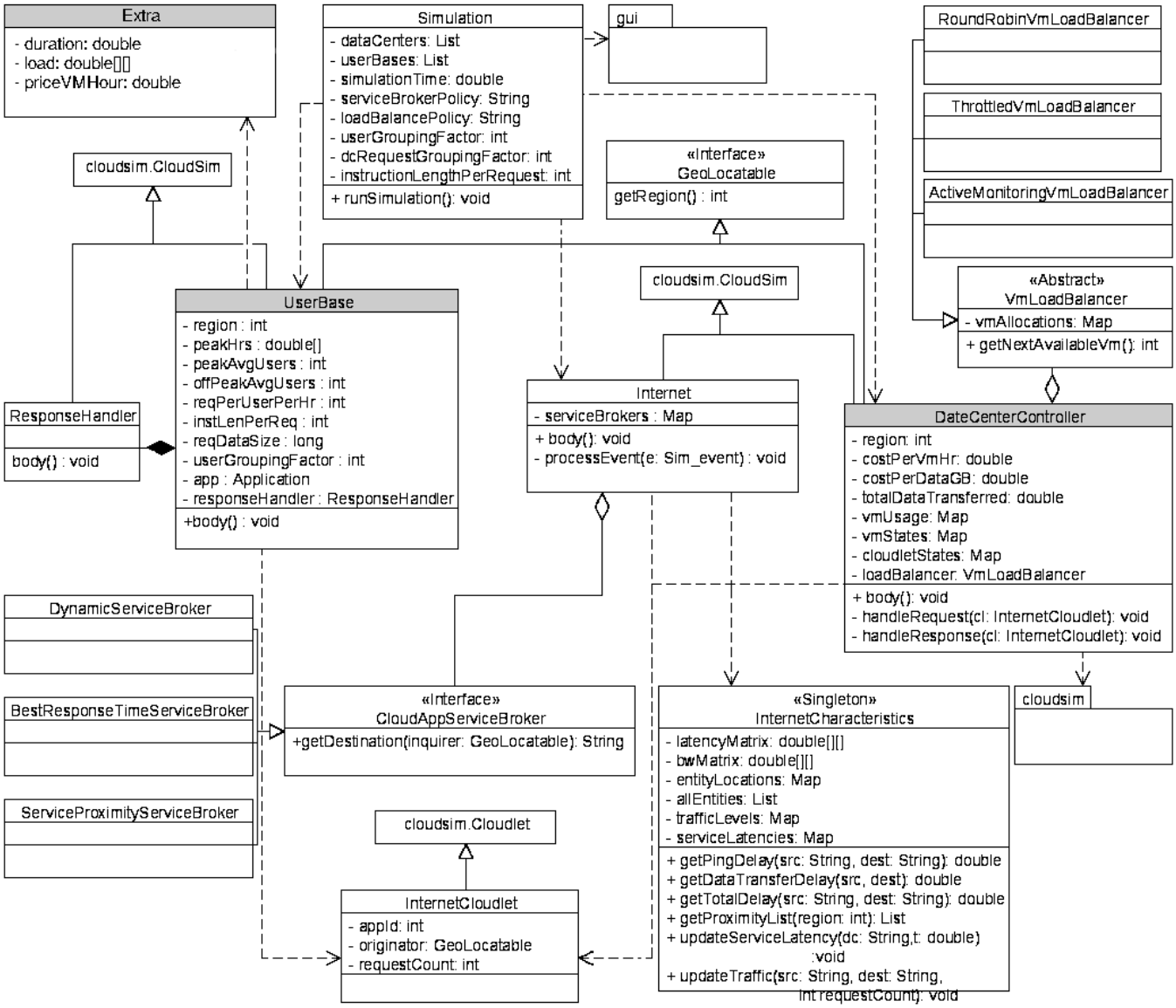


Figure 4.5: Class diagram of CloudAnalyst [1]

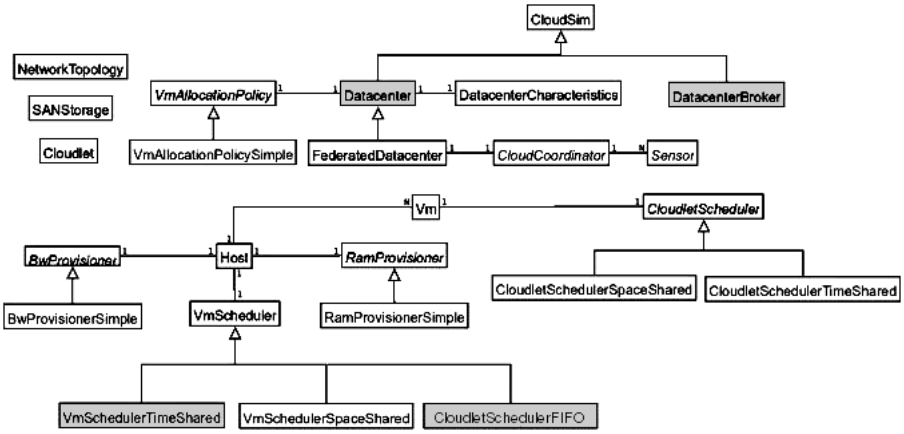


Figure 4.6: Class diagram of CloudSim [2]

parameter meanings are: The id of the sender of the message (5 in the case of DataCenterBroker), the id of the new VM we want to add, and a flag that differentiates a VM request message from its acknowledgment counterpart. With processMyVMCreate(), we go through the rest of the normal code path of the broker in processVMCreate(), which is normally triggered by an event. Here, we supply the message parameters in the 3-position vector.

```
public void addVMs(int number){
  long size = 10000; //image size (MB)
  int memory = 512; //vm memory (MB)
  ...
  int actual=vmlist.size();
  for (int i = 0; i < number; i++) {
    VirtualMachine aux = new VirtualMachine(new VMCharacteristics(
      actual+i, 6, size,memory, bw, vcpus, priority, vmm,
      new TimeSharedVMScheduler()));
    vmlist.add(aux);
    }
  int [] auxMap=this.vmMapping;
  this.vmMapping = new int[vmlist.size()];
  for(int i=0;i<auxMap.length;i++)
    vmMapping[i]=auxMap[i];
  for(int i=actual;i<this.vmlist.size();i++)
    vmMapping[i]=-1;
  processMyResourceCharacteristics(datacenterChar[0]);
  for(int i=actual;i<this.vmlist.size();i++){
    int[] array = new int[3];
    array[0]=5;
```

```
    array[1]=i;
    array[2]=1;
    processMyVMCreate(array);
}}
```

The code above is the result of trial and error due to lack of documentation. The sequence necessary was learned by adding a debugging function to the CloudSim class to trace the event exchange and observing the start of a simulation. The resulting output on the console showed a message from the DataCenter (id 6) to DataCenterBroker (id 5) with tag 1002, which is VM creation, containing a VMCharacteristics object, and the acknowledgment back:

```
Starting internet 9
5.0: DC1-Broker: Cloud Resource List received with 1 resource(s)
5.0: DC1-Broker: Trying to Create VM #0
Message-> destID:5 delay:0 gridSimTag:1002 this.Sim_id:6
  entityName: *No_entity* message:null
  data:cloudsim.VMCharacteristics@12ee2a destPort:null
Message-> destID:6 delay:0 gridSimTag:1002 this.Sim_id:5
  entityName: *No_entity* message:null data:[I@1880b02 destPort:null
```

4.2.1.2 VM Addition Testing

Having implemented VM addition, an experiment was devised to test it. It observes the simulated latency of requests coming at a fixed rate of 2000 requests an hour, while the number of VMs in the datacenter is rising. It starts with 20 VM and adds one each 5 minutes during one day. Internet Characteristics, Cloudlet complexity, and Host properties are left at their defaults. The number of Hosts in the datacenter was constant. Measurements are taken each 5 minutes. The data obtained from this simulation is in Figure 4.7

As seen on the figure, the average response time at the beginning with 20 machines is almost 1000 ms. It is falling as VMs are added. Once the number of VMs reaches about 160, the response time reaches its minimum, with a response time slightly higher than 100 ms. From this moment, anomalous behavior is observed. The more VMs are added, the higher the response time. In a practical system, it could be attributed to overhead at the load balancer or the cost of VM switching and CPU sharing, however, as far as we know, no such effects are simulated by CloudSim, so the expected curve would be constant once the minimum is reached.

As was learned later, when working on the queueing logic of CloudSim, the source of the anomaly likely lies in the way processor power of physical hosts is distributed among VMs. The queueing logic only works inside a VM, and to know how many instructions each VM can execute per time quantum, a value called mipsShare is computed by dividing the available power by the number of running VMs. It is computed in the class TimeSharedVMScheduler. That way, when adding VMs, once there are more VMs than physical processors, the computing power of one VM is getting lower.

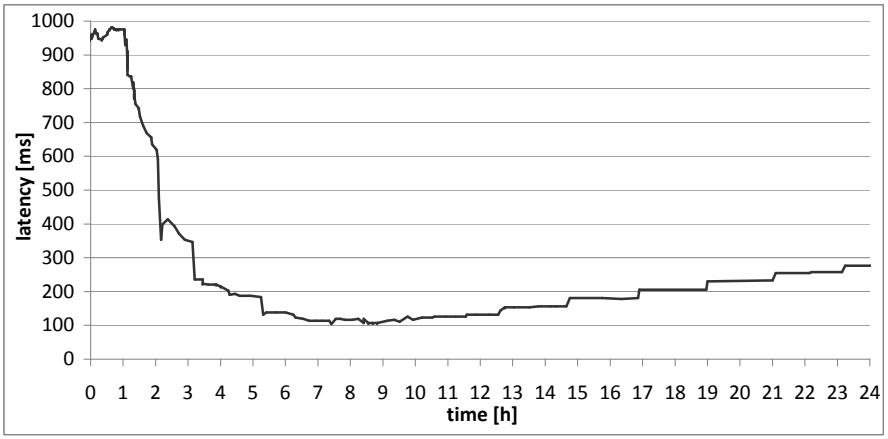


Figure 4.7: Simulated latency when adding VMs while keeping load constant

In reality, an idle VM on a hypervisor does not consume any processor cycles and a running VM on the processor will get its whole power (minus context switching overhead). As it is possible to query whether there are some cloudlets in a VM's run queue, it is conceivable to write more realistic VM schedulers. Until then, processor oversubscription in CloudSim may result in incorrect results. We believe that it applies to all versions.

It is possible that this inconsistency between simulation and reality may have affected other authors as well. The Figure 6 of the article [73] by Bessis et al. somewhat resembles Figure 4.7 here. The experiment is also very similar, as it adds VMs over the capacity of physical hosts, which results in an increase of service times. The authors state at the end of Section 4.1 of the article that a cloud typically has two levels of schedulers, which leads us to believe that they did not expect the outer level of the CloudSim scheduler to be static.

4.2.1.3 Autoscaler implementation

Afterwards, VM deletion was implemented in a similar way to addition and a simple autoscaler was implemented. Instead of adding a VM every 5 minutes, the function checks the 5-minute average response time at a Datacenter. It compares the value to an upscaling and a downscaling threshold and adds/removes one VM based on the result. An approximation of a daily load curve was input and the thresholds were set to 330 ms and 190 ms. No VM set-up time was simulated, but it could be done by simply not counting the last 5 minutes of the statistics. Figure 4.8 shows the number of request an hour simulated during the day.

When the Datacenter contained a constant 20 VMs, the response time followed the load, as seen in Figure 4.9. The spikes at the beginning of each hour are a mainly due to high variance of the incoming request flow due to a design flaw in UserBase code, as was learned later and explained near the end of Section 3.2.2.

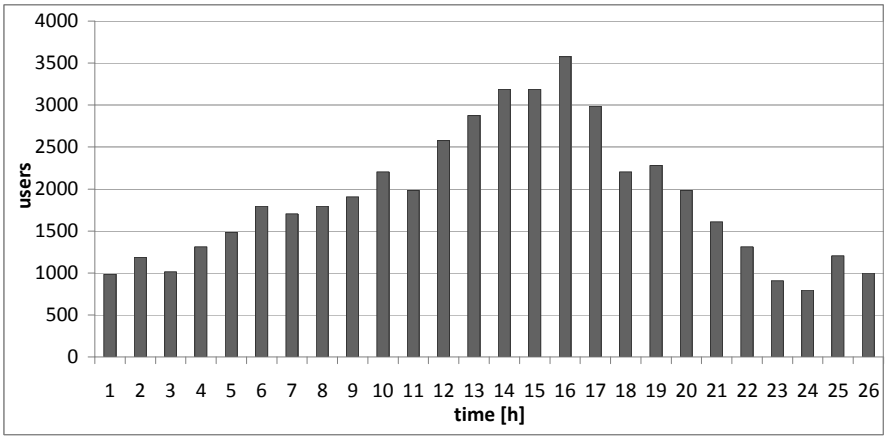


Figure 4.8: Simulated daily load curve, amount in requests per hour

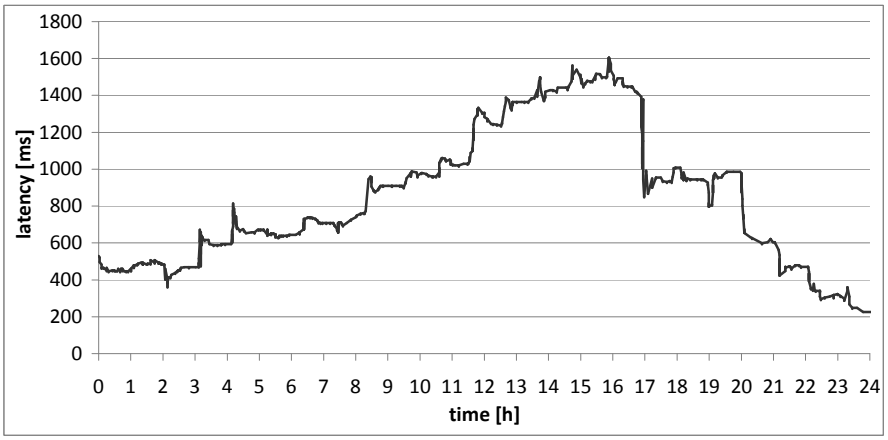


Figure 4.9: Simulated latency with a constant number of VMs

According to the statistics, the processing time was 949 ms on average, with a minimum of 110 ms and a maximum of 2094 ms. If the threshold function described above is applied to the same input, the result on Figure 4.10 is received.

As we can see, the latency plot begins at the same point as in the previous figure, but the scaling function immediately starts adding VMs to keep the latency inside the limits. Also, we may observe a decreasing slope after the latency rises over 330 ms at hour 12 and an increasing slope when VMs are being removed after hour 16. The average response time has decreased down to 280 ms, the minimum is still at 110ms and the maximum has gone down to 700 ms, which lies at the start of the simulation, where the parameters were outside the autoscaler's control. On average, the processing time decreased by 70% from the static case.

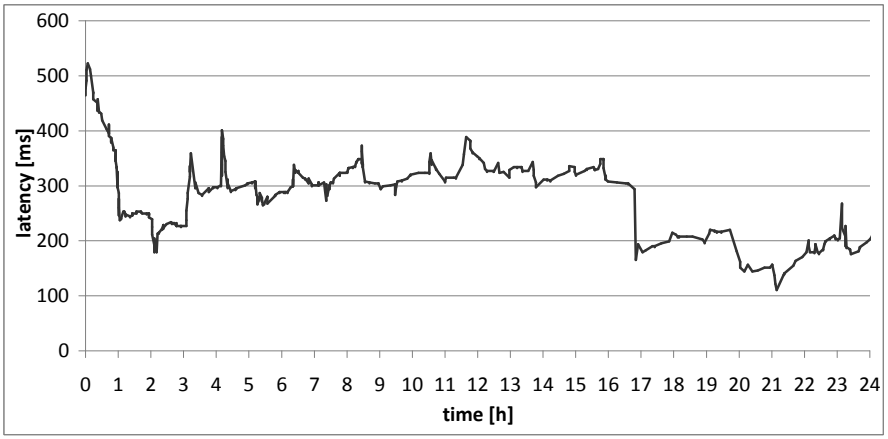


Figure 4.10: Simulated latency with automatic scaling

4.2.2 Modifications to increase accuracy

Seeing these discrepancies, we decided to delve deeper into the queueing code of CloudSim. The most important class, and the only one where Cloudlets are "computed", *i.e.* their instruction counts decreased based on time and allocated processing power, is CloudletSchedulerTimeShared, called TimeSharedVMScheduler in version 1. The return values from the two most important methods, cloudletSubmit() and updateVMprocessing() are expected finish times of the next cloudlet, so that the firing time for an event in the discrete event simulation engine can be set.

Browsing through the CloudSim changelogs[1] to see whether there have been any bugs fixed, we have seen that there were at least 12 changes, some referencing precision fixes and rounding issues. The changelog does not even go back to version 1 beta. Indeed, the code seemed to count instruction counts multiple times in cases when there was more than one cloudlet in execution, making the system much more loaded than it should be.

Because, as stated earlier, the new version of CloudSim lacks features we used to implement the instrumentation necessary for autoscaling, we decided to backport the queueing logic to CloudAnalyst. It was mostly compatible. There were some type changes and the rather annoying need to edit every occurrence of the class Cloudlet back to Gridlet. A significant change was that cloudletSubmit() previously returned available capacity and the time of the next event was computed elsewhere. The new version returns the time directly. We backported that change as well for consistency.

Unfortunately, just porting the code did not suffice. We found the computations were off by several orders. After a long investigation of the code of CloudAnalyst and CloudSim 3, we found that the new version counts time in seconds and the minimum time between events is 0.1 by default. (This number is intentionally left without units to illustrate how

[1] `https://code.google.com/p/cloudsim/source/list?path=/trunk/modules/cloudsim/src/main/java/org/cloudbus/cloudsim/CloudletSchedulerTimeShared.java`

hard it is to orient yourself in code where no quantity has documented units, except perhaps MIPS, which you can use to orient yourself.) CloudAnalyst uses milliseconds and has no minimum. This is consistent with CloudSim being designed for simulation of batch jobs and CloudAnalyst for interactive services. We had to adapt all formulas in the backported class to reflect this. A side effect is that in order to fix the broken CloudSim, we had to put its processing power in Gips (German readers, forgive the pun).

A big problem of CloudSim in both examined versions is rounding errors. As time is stored as a double precision floating point variable, zero sometimes is not exactly zero. Mainly when there is a small fraction of a Cloudlet left to process, this fraction is divided by available MIPS and the next event scheduled based on that value, such as:

$$T_{next\_event} = T_{now} + \frac{MI_{left}}{MIPS_{SHARE}}$$

An underflow occurs and in the next time interval, the cloudlet is again not finished. This not only makes the simulation take longer because of excessive events, but also does weird things to Cloudlet durations. To be more concrete, we think it extends their duration an unknown number of times by a time interval that is about one to three orders of magnitude (depending on the MIPS value) above the minimum number that can be represented by a Java double on a particular Java implementation.

Solving this by introducing a minimum time between events is in our opinion wrong, because the time-shared scheduler will first compute the mipsShare of running cloudlets and then run the time quantum. A cloudlet that was supposed to finish will not only continue running up to minTimeBetweenEvents longer, but will also increase the runtime of all other concurrent cloudlets, because less MIPS will be available to the for the one quantum. If a lot of cloudlets are running on a particular VM, this could become a problem. We solved it by forcefully finishing cloudlets that have estimated remaining time less than 0.1 ms. They are also immediately removed from the queue and the mipsShare recomputed. Depending on the length of cloudlets processed, the reader is encouraged to set it lower, but not above the threshold, where rounding errors may appear. In our case of cloudlets running over 100 ms, this introduces an error of at most 0.1%.

To get rid of all errors with rounding and concurrent execution, we have also implemented a reference FIFO queueing discipline, which in our opinion (perhaps biased by study of queueing theory), should have been there from the start, instead of the PS (Processor Sharing) CloudletScheduler. It runs the Cloudlets one by one without any concerns about mipsShares, and correctly computes the run time of the whole queue on cloudletSubmit(). With the removal of near-finished Cloudlets to fix rounding errors, it computes the finish times precisely, and events are scheduled exactly to the end of the next CloudLet, which runs to completion in one time step, unless interrupted by arrival of a new request from a UserBase. The event flow is very clean. The modified PS queueing discipline gives nearly identical results to FIFO in our experiment.

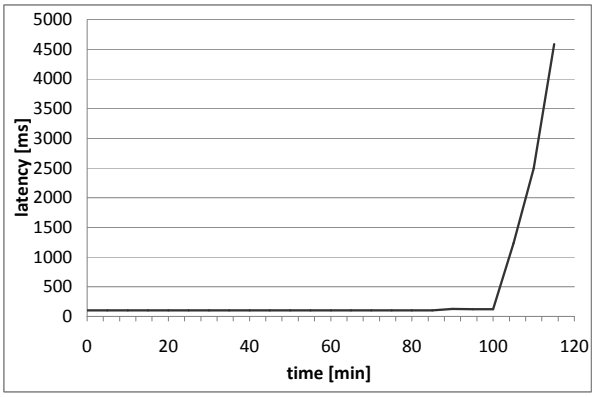


Figure 4.11: Simulation results from modified CloudSim

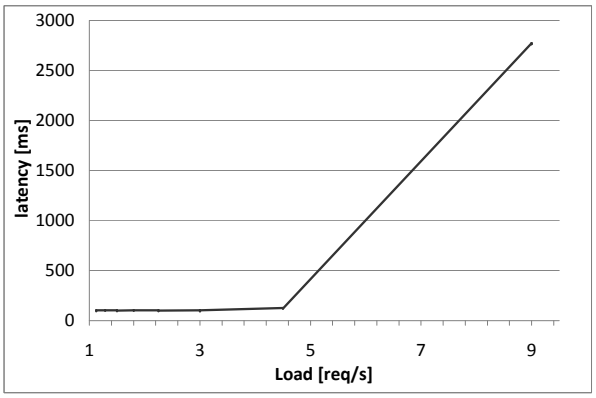


Figure 4.12: Result from modified CloudSim plotted against load

4.2.2.1 Load generator rewrite

After the last two modifications, CloudSim was already giving correct average latencies, but when writing more granular output to the terminal, we noticed large spikes of incoming requests. Looking closer at the load generator in the class UserBase, we noticed that when the user count is set to 1012 and requests per user per hour to 32, it does not generate a Poisson process of about 9 req/s, as we expected, but rather sends 1012 requests at once 32 times an hour (randomized by Poisson distribution). We fixed the load generator to work as expected. As a final addition, we also randomized the length of submitted Cloudlets to get from uniform service time to a M/M/N queue.

Later, we have found a technical report by the CloudAnalyst author [1], which describes its design in more depth than the conference article. The grouping of requests was apparently added to increase the speed of the simulation, but in doing so, the author changed the request arrival model from random with exponential interarrival times, which is usually employed to simulate a group of independent users, to batched, which models thousands of users, who click their mouse in unison. It will produce the same average latencies for hourly reporting, but is not usable for higher granularities, which we needed in order to simulate autoscaling.

The results on the preceding Figures 4.11 and 4.12 were obtained from the modified CloudSim with parameters Cloudlet complexity 715 MI, processor power 7 GIPS, incoming load of 1086 users generating 32 req/h, that is 9.65 req/s, number of processors again decreasing from 8 to 1.

Table 4.6 summarizes the results of the experiments. The first line contains the result of the real load test. Following are results from CloudSim and their errors from the load test, in the following order: The unmodified version with the original queueing logic, the fixed version and the version with added randomization. As can be seen, the randomization of service times is very important for correct results. For comparison, the results from the PDQ queueing network model shown in Figure 3.8 are also presented in table form. To

Table 4.6: Simulation and Load Test Results

Step	1	2	3	4	5	6	7	8
Real	108.41	106.12	105.14	104.42	103.86	106.60	127.54	3181.2
Original	87.64	110.07	139.31	180.70	242.43	346.05	552.12	1171.4
error[%]	-19.15	3.72	32.49	73.05	133.41	224.60	332.89	171.56
Fixed	102.46	102.46	102.46	102.46	102.46	102.46	116.84	281.34
error[%]	-5.48	-3.44	-2.55	-1.87	-1.35	-3.88	-8.38	1030.7
Random	102.26	102.23	102.34	102.41	102.32	102.46	123.15	2771.9
error[%]	-5.67	-3.66	-2.66	-1.92	-1.47	-3.88	-3.44	14.76
PDQ	102.14	102.14	102.15	102.22	102.75	106.35	133.37	3172.5
error[%]	6.14	3.89	2.93	2.15	1.08	0.24	-4.36	-0.27

get this close to the load test, the incoming flow was parameterized at 9.45 req/s. Service demand is the same as in CloudSim. Results from both methods are sufficiently close to reality for practical applications. (Once again, the first steps of the real experiment are burdened with additional overhead from hyperthreading and VM context switching.)

4.3 Cloud Simulator based on Queueing Theory

We propose a new simulator of cloud automatic scaling that is a much improved version of the loop calling the PDQ QN model, which was used to produce the last line of Table 4.6.

To demonstrate how a user would use our simulator to tune an autoscaler to their application, we will present its results when implementing standard threshold-based algorithms with different threshold settings. All three threshold-based autoscaling strategies (utilization, latency, and queue length based) will be implemented.

4.3.1 Definitions of data and metrics

The data used for the evaluation is the same as in the previous Section 4.1. The data came from a small Czech web hosting company and was collected in nearly three months starting in the period around Christmas, meaning there are both areas of normal traffic and anomalous holiday traffic.

In this experiment, we will use the series oe, which is a large web shop. It has a clean curve of traffic with nearly 0 req/s at night, around 100 req/s on normal days and spikes up to 300 req/s. Its character is a prototype of an interactive business site. The trace collected is of CPU load while the simulator needs request intensities. However, we know that the server is never overloaded, so the dependency of load on intensity is linear. Therefore, the character of the traces is the same. The exact values are irrelevant because we will create a what-if scenario of 16 times the traffic to induce the need for scaling of the number of servers. Otherwise, the simulation of one underloaded server would not be very interesting. The service demand of 30.6 ms was calculated from the Utilization Law, which states that

the average utilization of individual servers equals the request throughput multiplied by the service demand, divided by the number of servers

$$U = \frac{XS}{N}$$

The necessary values were taken recently from weekly averages of typical weeks without traffic spikes. The experiment will use the period of 14 days from 24.12.2012, which is the least busy day of the year. On the other hand, on 3.1.2013, there is a big spike in demand, probably caused by a discount after Christmas.

The second time series used will be bender, which is a shared web hosting. It also has a visible daily curve, but with significantly more noise, as the level is usually only 5 req/s with 10 req/s spikes. It was also never overloaded in the measurement period. A weekly average response time of 150 ms will be used as service demand and the traffic intensity will be amplified 16 times. The chosen time interval is two weeks starting 1.1.2013, as the holiday traffic was uninteresting and demand rose after the start of the first work week.

The last series used will be gaff, which is a web shop aggregator and search engine. Its load curve is atypical because it is dominated by long-running spikes of data imports, but it has proven to be predictable using time series analysis methods. The service has been discontinued, so we do not have access to more information than the collected CPU load trace. The service demand will be set to 35 ms and amplification to 8. The most interesting period seems to be 14 days ending at 24.12.2012 because there is some interactive search traffic besides the spikes caused by scheduled jobs.

To evaluate the efficiency of different autoscaling strategies, we define two cost metrics. The first will be the cost of running the cloud service expressed in machine-hours. The second will be the number of virtual machine boots per the observation period. It is the same as the number of shutdowns, or differs by a very small number, because the measurement period is aligned to start and end at the same time of day.

As the tuning of an autoscaler is a compromise between cost and performance, we will measure the fulfillment of an SLO (Service Level Objective). A modern way to quantify user satisfaction with an interactive service is Apdex [85][1]. It is a simple formula which operates on response times. Users are divided into three groups - satisfied, tolerating, and frustrated, using two response time thresholds, T and F. T is chosen by the user and

$$F = 4T$$

The resulting index is computed as the ratio of satisfied and tolerating requests to the whole, where tolerating are counted as half, such as:

$$Apdex = \frac{Req_{satisfied} + \frac{Req_{tolerating}}{2}}{Req_{all}}$$

As to the setting of the T threshold, the specification does not say anything. The practitioners of the method are divided into two camps. The stricter one is represented by New Relic[2] and says that the threshold is to be set close above the average response time,

[1] `http://apdex.org/index.php/alliance/specifications/`

[2] `http://blog.newrelic.com/2011/01/21/what-should-i-set-my-apdex-goal-to/`

to better capture any fluctuations and performance problems. The more conservative one is represented, e.g., by Coscale[1] and proposes to set it between 1 and 3 seconds, which is the time when the average web user will actually begin to get frustrated by the response time. We will include both and label them ApdexS for strict and ApdexC for conservative. ApdexS T threshold will be set to the service demand, rounded up on its first significant digit. ApdexC T threshold will be set to 1 s.

We will not be computing the values on individual requests because the model does not see them, but on whole time intervals, because the result of the model is the average service time. However, there is more information available – the resulting service time is exponentially distributed in case of a single server and is governed by the Erlang C formula in case of multiple servers.

Inspired by the article by PDQ's author[2], which explains how to add confidence intervals to PDQ output, we will use the Erlang C formula to extend the model. The formula itself calculates the probability of waiting $p(>0)$ in a multi-server open queueing system as function of the utilization A (in Erlangs, $A = NU$, that is utilization of a single server times the number of servers) and number of servers N. It serves as the basis of other calculations on these systems, most notably the average waiting time and the service level (or GoS). The service level here is a probability that a request will wait for less than a specified time threshold. Besides the inputs of Erlang C, it depends on the service rate μ. The GoS formula is:

$$P(T_w \leq t) = 1 - p(>0)e^{-(N-A)\mu t}$$

An example of a manual calculation of GoS is found in a web page by Tanner[3]. The Erlang C formula itself was taken from the "queueing" library for R [86].

This well-known formula only deals with the time a request spends in the queue, which is not what we need to calculate the Apdex value from the model. It was designed for call center dimensioning, where a customer is satisfied when an operator answers his or her call. To compute the time when a web user begins to see the results of a request, we must consider the sum of waiting and service time. The formula to calculate a point in its distribution function is:

$$A \neq N - 1 \tag{4.1}$$

$$\mathbf{P}\{D > t\}_{FIFO} = \frac{p(>0)}{A + 1 - N}[e^{-(N-A)\mu t} - e^{-\mu t}] + e^{-\mu t} \tag{4.2}$$

$$A = N - 1 \tag{4.3}$$

$$\mathbf{P}\{D > t\}_{FIFO} = (\mu t\, p(>0) + 1)e^{-\mu t} \tag{4.4}$$

$$\mathbf{P}\{D \leq t\} = 1 - \mathbf{P}\{D > t\} \tag{4.5}$$

Calculation of the average Apdex value for a time interval from this is straightforward. As a performance metric of autoscaling algorithms, we will count the absolute number of

[1] http://www.coscale.com/blog/web-application-performance-what-apdex-doesnt-tell-you

[2] http://perfdynamics.blogspot.nl/2013/04/adding-percentiles-to-pdq.html

[3] http://www.mitan.co.uk/erlang/elgcmath.htm

time intervals, when ApdexC was below 0.95 (and users were seeing significant latencies) and the percentage of intervals when ApdexS was below 0.7 (and the system was overloaded from a performance engineering standpoint).

In the two-tier version of the simulator, we also use this formula. The distribution function that it represents is sampled for each tier and a fast linear convolution function from R "stats" library is used to obtain the distribution of the sum of the two random delays.

With the performance metrics defined, we conducted a parameter sweep experiment to evaluate the three different threshold-based algorithms on our three example series. The input parameters were the upscaling and the downscaling threshold, within reasonable bounds for the respective workloads.

The resulting tables of the output metrics have been analyzed by hand to identify acceptable bounds for the metrics. The tables have been marked with bold font for good values, normal for acceptable and italic for bad. The Apdex metrics' thresholds for this marking are set uniformly for all three series to 5 and 10 percent for Apdex S below 0.7 and 10 and 25 occurrences for ApdexC below 0.95. The machine hours and start-stops are set individually based on the results of each experiment. The resulting tables can be used to select a right threshold combination for a particular workload by quick visual inspection.

4.3.2 The utilization-based autoscaler

The first evaluated algorithm will be the most standard one used in most IaaS clouds. It is also referred to throughout this work as the Amazon Web Services type.

For illustration, in Table 4.7 we can see the results of the parameter sweep on the utilization-based autoscaler working on the series oe with different thresholds. The up threshold was varied from 50 to 90 and the down threshold from 10 to the up threshold minus 10. Due to the smooth nature of the time series, we have a good example of what the result table should look like. With lower utilizations, the number of used machine-hours tends to be higher, and as the machines are utilized more, there is a greater risk of SLO violations. The number of starts/stops is nearly constant across the table, except when the thresholds were too close together or in cases, where the down threshold was 50 and more, which caused the system to oscillate. Larger values have been trimmed from the table. The starts/stops metric seems to serve as a good indication of unstable autoscaler parameters.

If the gentle reader agrees with our bold/italic marking of the table, then it is evident that 70/20 or 80/20 would be good choices of parameters for this workload. We observe that the biggest strength of the utilization-based autoscalers is that these choices are more or less universal. The user will not make a great mistake when he or she applies them to any workload. This is probably due to utilization being a derived metric of the workload intensity. The latency is also closely tied to it. With workloads of this type, it is important to keep the system in light load mode with minimal queueing. It is important to remember the utilization-latency characteristic of the M/M/N queueing system we are working with

Table 4.7: Utilization-based autoscalers on "oe"

up	down	hours	starts	stops	ApdexC	ApdexS
50	10	*1240*	**69**	**62**	**0**	**1.04**
50	20	*1162*	**73**	**70**	**0**	**1.26**
50	30	*1116*	88	87	**0**	**2.01**
50	40	*1072*	*132*	*131*	**1**	**2.6**
60	10	*1084*	**57**	**51**	**0**	**2.9**
60	20	*1017*	**58**	**55**	**0**	**3.12**
60	30	980	**64**	**63**	**0**	**4.01**
60	40	946	83	82	**1**	**4.68**
60	50	938	*144*	*143*	**5**	5.28
70	10	980	**50**	**45**	**0**	**4.61**
70	**20**	918	**51**	**48**	**0**	**4.83**
70	30	**885**	**55**	**54**	**0**	6.1
70	40	**856**	**66**	**65**	**2**	7.51
70	50	**852**	*103*	*102*	8	8.55
80	10	**899**	**45**	**40**	**1**	8.85
80	*20*	**844**	**46**	**43**	**1**	9.07
80	30	**817**	**48**	**47**	**1**	*11.08*
80	40	**790**	**51**	**50**	**3**	*13.01*
80	50	**799**	87	86	9	*14.35*
90	10	*1015*	**49**	**45**	10	*13.75*
90	20	921	**49**	**46**	10	*14.05*
90	30	**835**	**50**	**49**	11	*16.21*
90	40	**775**	**59**	**58**	14	*19.93*
90	50	**769**	84	84	21	*23.49*

Table 4.8: Utilization-based autoscalers on "bender"

up	down	hours	starts	stops	ApdexC	ApdexS
50	10	*1211*	**16**	**14**	**0**	**0.15**
50	20	948	**62**	**61**	8	**1.34**
50	30	858	*154*	*154*	*27*	**4.46**
50	40	847	*245*	*244*	*70*	8.18
60	10	*1077*	**9**	**7**	**0**	**0.45**
60	20	840	**44**	**43**	15	**2.68**
60	30	761	*111*	*110*	*42*	7.73
60	40	**749**	*199*	*198*	*91*	*12.12*
60	50	783	*263*	*262*	*121*	*14.8*
70	10	987	**7**	**6**	**3**	**0.89**
70	*20*	791	**34**	**33**	*26*	**4.83**
70	30	**720**	94	93	*75*	*12.57*
70	40	**697**	*163*	*163*	*137*	*18.59*
70	50	755	*250*	*249*	*168*	*20.67*
80	**10**	968	**7**	**6**	**5**	**1.19**
80	20	765	**35**	**34**	*36*	6.17
80	30	**690**	94	93	*127*	*19.11*
80	40	**700**	*162*	*161*	*195*	*25.8*
80	50	759	*248*	*247*	*229*	*28.48*
90	10	878	**6**	**5**	9	**2.75**
90	20	759	**40**	**39**	*75*	*10.26*
90	30	**718**	*119*	*119*	*203*	*25.06*
90	40	**725**	*175*	*175*	*275*	*32.34*
90	50	799	*267*	*267*	*323*	*35.32*

(The first example we could find is again by the author of PDQ[1], but this characteristic is well known.). With more servers, the knee between the light load and heavy load mode moves to higher values. The mentioned thresholds may very well stop working if we go beyond eight servers. The utilization-based autoscaling model will become harder to tune right the more servers we add.

Looking at the results for time series bender, we note that the model is much more sensitive to changes in the autoscaling thresholds, particularly the down one. Models with the threshold above 20 exhibit many oscillations, probably due to the noisy nature of the data. With the smoother series oe, this effect manifested itself above 50. Still, the combination 70/20 would be viable. 80/10 could be used if we had strict requirements on ApdexC violations and were willing to pay more for computing costs.

[1] `http://perfdynamics.blogspot.nl/2009/07/remembering-mr-erlang-as-unit_29.html`

The results on series gaff clearly show that while using reactive threshold-based autoscalers, it is impossible to match the abrupt changes in the level of the series. The rising trend of SLO violations against falling machine hours is still present, although it is not as pronounced as with the other series. The values of choice would likely be 70/50 or 70/10.

4.3.3 The latency-based autoscaler

The second evaluated algorithm is found in some PaaS clouds. It is also referred to as the Google App Engine type.

Table 4.9: Latency based autoscalers on "oe"

up	down	hours	starts	stops	ApdexC	ApdexS
40	30	*1612*	**5**	**0**	**0**	**0**
40	32	972	91	90	**0**	**0.07**
40	34	956	*163*	*162*	**0**	**1.56**
40	36	949	*196*	*195*	**0**	**3.72**
40	38	944	*208*	*208*	**0**	**4.83**
50	30	*1417*	**5**	**0**	**0**	**0**
50	32	**878**	**56**	**55**	**0**	**0.74**
50	34	**853**	85	84	**0**	**1.86**
50	36	**847**	*121*	*120*	**0**	**4.01**
50	38	**840**	*152*	*152*	**2**	6.32
50	40	**836**	*165*	*165*	13	7.51
60	30	*1360*	**5**	**0**	**0**	**0.37**
60	32	**829**	**45**	**44**	**0**	**3.35**
60	**34**	**807**	**66**	**65**	**0**	**4.83**
60	36	**801**	91	90	**0**	6.62
60	38	**800**	*116*	*116*	**2**	8.48
60	40	**795**	*137*	*137*	15	*10.04*
70	30	*1305*	**4**	**0**	**1**	**0.67**
70	32	**801**	**42**	**41**	**1**	6.77
70	*34*	**773**	**50**	**49**	**2**	8.33
70	36	**769**	**74**	**73**	**2**	*10.26*
70	38	**769**	96	96	**3**	*11.97*
70	40	**768**	*114*	*114*	14	*13.09*

Table 4.10: Latency-based autoscalers on "bender"

up	down	hours	starts	stops	ApdexC	ApdexS
300	150	*1112*	**3**	**0**	**1**	**0.3**
300	**152**	773	**15**	**14**	**5**	**1.19**
300	154	752	**22**	**21**	6	**1.64**
300	156	**716**	**40**	**39**	13	**2.9**
300	158	**702**	**55**	**54**	21	**4.01**
300	160	**694**	**70**	**69**	*27*	5.06
300	162	**688**	81	81	*28*	5.87
300	164	**684**	91	91	*35*	6.77
500	150	*1080*	**3**	**0**	**5**	**0.97**
500	152	**724**	**7**	**6**	10	**2.53**
500	154	**705**	**14**	**13**	19	**4.39**
500	156	**672**	**23**	**22**	*32*	6.25
500	158	**656**	**33**	**32**	*45*	9.22
500	160	**642**	**45**	**44**	*63*	*11.82*
500	162	**633**	**55**	**54**	*74*	*13.75*
500	164	**631**	**63**	**62**	*84*	*14.13*
700	150	*1029*	**3**	**0**	12	**2.38**
700	*152*	**697**	**5**	**4**	15	**3.79**
700	154	**677**	**12**	**11**	*29*	6.25
700	156	**656**	**19**	**18**	*46*	8.25
700	158	**635**	**27**	**26**	*68*	*12.49*
700	160	**617**	**35**	**34**	*94*	*16.06*
700	162	**609**	**42**	**41**	*108*	*18.29*
700	164	**603**	**48**	**47**	*125*	*19.85*

The character of the result table of latency-based autoscalers on the series oe is a little different from the utilization-based case. We do not see a clear trend throughout the table, rather the cost decreases and SLO violations increase in every subseries characterized by a different up threshold. Also, when the up threshold is low, there is a tendency for

oscillations. Higher up thresholds tend to have lower cost and cause lower performance. The values of choice would be 60/34 or 70/34 ms. Compared to the utilization-based autoscaler, these settings mean significantly less used machine hours with the same Apdex scores – 807 and 773 versus 918 and 844 hours.

The character of the result table for time series bender is very similar to the previous one. However, the change from the utilization-based case is different. We can see a lower number of extreme values of both starts/stops and hard SLO violations (ApdexC), while the ApdexS values are in the same range. Also, the number of machine-hours used tends to be lower. The values of choice would be 700/152, or 300/152 ms for minimal ApdexC violations. Compared to the choices mentioned for the utilization-based autoscaler, the user would again save computing power – 697 and 773 versus 791 and 968 hours. In the case with lower performance requirements, the number of machine starts and stops is also significantly lower.

Table 4.11: Utilization-based autoscalers on "gaff"

up	down	hours	starts	stops	ApdexC	ApdexS
50	10	658	83	83	14	**2.75**
50	20	**569**	*103*	*103*	17	5.43
50	30	**566**	*107*	*107*	17	5.8
50	40	**569**	*112*	*112*	18	5.87
60	10	628	**70**	**70**	14	**2.83**
60	20	**531**	85	85	17	6.62
60	30	**529**	89	89	17	6.99
60	40	**524**	90	90	17	7.14
60	50	**511**	92	92	17	7.66
70	**10**	618	**67**	**67**	16	**3.87**
70	20	**514**	77	77	17	7.81
70	30	**511**	80	80	17	8.62
70	40	**508**	80	80	17	8.7
70	*50*	**491**	80	80	17	9.07
80	10	619	**69**	**69**	18	**4.16**
80	20	**515**	78	78	20	8.77
80	30	**510**	79	79	21	9.81
80	40	**507**	79	79	21	9.81
80	50	**488**	79	79	21	*10.26*
90	10	619	**69**	**69**	18	**4.16**
90	20	**523**	82	82	22	9.07
90	30	**518**	83	83	23	*10.11*
90	40	**515**	83	83	23	*10.11*
90	50	**496**	83	83	23	*10.56*

Table 4.12: Latency-based autoscalers on "gaff"

up	down	hours	starts	stops	ApdexC	ApdexS
50	35	*1341*	**3**	**0**	**2**	**0.22**
50	**36**	**556**	100	100	19	**3.2**
50	37	**553**	*122*	*122*	17	**3.79**
50	38	**551**	*124*	*124*	19	**3.87**
50	39	**555**	*128*	*128*	19	**4.31**
50	40	**562**	*131*	*131*	21	**4.39**
50	41	**564**	*135*	*135*	21	**4.91**
50	42	**573**	*137*	*137*	21	5.2
60	35	*1341*	**3**	**0**	**2**	**0.22**
60	36	**509**	78	78	*32*	5.06
60	37	**500**	92	92	*31*	5.95
60	38	**499**	92	92	*31*	5.95
60	39	**498**	94	94	*31*	6.25
60	40	**498**	97	97	*31*	6.32
60	41	**499**	*101*	*101*	*32*	6.54
60	42	**506**	*103*	*103*	*32*	6.69
70	35	*1341*	**3**	**0**	**2**	**0.22**
70	36	**491**	**68**	**68**	*36*	6.84
70	37	**481**	77	77	*36*	8.1
70	38	**480**	77	77	*36*	8.1
70	39	**475**	78	78	*36*	8.62
70	40	**473**	78	78	*36*	8.62
70	41	**472**	80	80	*37*	8.77
70	42	**476**	82	82	*38*	8.92

With the series gaff, the latency-based autoscaler exhibits similar characteristics to oe, that is instability with low up thresholds and a tendency for a higher number of VM starts. Compared to the utilization-based autoscaler, it can achieve slightly lower counts of machine-hours with similar start/stop counts and soft SLO violations (ApdexS), but at the cost of a higher number of hard SLO violations. It is very difficult to select a parameter combination that would satisfy all our criteria. 50/36 with its 556 machine-hours beats the utilization-based autoscaler second variant with 618 h, at the cost of more start/stops 100 vs. 67.

It would seem that the latency-based autoscaler is better, as we were able to point out parameters with lower cost for every time series analyzed. However, it also has its downsides, mainly greater sensitivity to the threshold settings. Where the utilization-based autoscaler has a near-universal working point that is dependent only on the number of servers, setting the latency thresholds requires the user to have sound knowledge of the resource demands of the application being scaled.

Please note that the down thresholds are very close to the service demand and that the stops value for the lowest down thresholds in every table is always 0. That is because the down threshold was set below the service demand and was never triggered. (The model then accidentally calculated a benchmark value of the maximum possible number of machine-hours used when VMs are never removed. The number of starts plus one (the starting value) equals the number of VMs needed at peak traffic intensity with the specified up threshold as SLO.) The situation when the service demand would rise above the set down threshold of the latency-based autoscaler could happen if there was background load on the cloud that would cause the perceived service demand of the application on the cloud platform to rise. The result would be a non-working autoscaler and wasted money.

On the other hand, if the average resource demand were to fall, the down threshold would suddenly be too high, and, based on the experiment, the margin for error is in single milliseconds. The result would be more oscillations and SLO violations. The possible causes of this situation could be a new, optimized version of the application. This means that autoscaler tuning would have to be integrated into the application testing process. Another approach is shown by Spinner et.al. [87], who train an on-line model of the application service demand.

Another cause could be a change of the workload mix in the case of an application that serves different types of content. In this simulation, we are taking a trace and working with its average service demand. In reality, the measured servers offer both static and dynamic content and therefore, the distribution of service demands is multimodal. If the mix of the modes (i.e. the ratio of served web pages and pictures) were to change, the latency-based autoscaler would probably break. In this light, we only recommend using a latency-based autoscaler to web services that offer a single type of content.

4.3.4 The queue length-based autoscaler

The third evaluated algorithm is found quite rarely in some PaaS clouds. It is also referred to as the Red Hat OpenShift type.

Table 4.13: Queue length-based autoscalers on "oe"

up	down	hours	starts	stops	ApdexC	ApdexS
2	0.13	*1871*	**54**	**47**	**0**	**0.74**
2	0.25	*1541*	87	80	**0**	**3.42**
2	0.38	*1488*	87	80	**0**	**3.42**
2	0.5	*1444*	87	80	**0**	**3.49**
2	0.63	*1406*	91	84	**0**	**4.01**
2	0.75	*1377*	95	88	**1**	**4.61**
2	0.88	*1359*	96	90	**0**	**4.54**
2	1	*1342*	97	93	**1**	**4.98**
4	0.13	*1180*	**37**	**30**	**0**	6.62
4	0.25	991	**51**	**44**	**0**	7.88
4	0.38	960	**51**	**44**	**0**	7.96
4	0.5	936	**51**	**44**	**0**	8.03
4	*0.63*	913	**53**	**46**	**0**	9.52
4	0.75	**892**	**56**	**49**	**1**	*10.56*
4	0.88	**882**	**56**	**50**	**1**	*10.86*
4	1	**876**	**57**	**53**	**1**	*11.08*

Table 4.14: Queue length-based autoscalers on "bender"

up	down	hours	starts	stops	ApdexC	ApdexS
2	0.13	*1880*	**7**	**0**	**2**	**0.59**
2	0.25	*1826*	**8**	**1**	**2**	**0.59**
2	0.38	*1397*	**21**	**19**	8	**1.56**
2	*0.5*	972	**55**	**53**	*31*	5.13
2.5	0.13	*1684*	**7**	**0**	**3**	**0.74**
2.5	0.25	*1631*	**8**	**1**	**3**	**0.74**
2.5	0.38	*1201*	**17**	**16**	11	**2.01**
2.5	0.5	855	**42**	**41**	*44*	7.81
3	0.13	*1462*	**5**	**0**	**3**	**0.74**
3	0.25	*1411*	**6**	**1**	**3**	**0.74**
3	0.38	*1063*	**15**	**14**	12	**2.08**
3	0.5	810	**38**	**37**	*50*	8.55

The last parameter calculated by the queueing network model is average queue length. Let us see, what happens, when a threshold-based autoscaling algorithm is used on this parameter. The result table for series oe looks similar in character to the utilization-based autoscaler on the same series, having a single trend in the hours and Apdex columns, with the exception that the values for used machine-hours are higher for the same ApdexS. Therefore, it is not possible to select a combination of parameters that would have acceptable hours and start/stops metrics with ApdexS violations lower than 5%. Even the combination 4/0.63 results in somewhat higher cost (913 vs. 844 h). Compared to the latency-based case, it cannot compete in cost or achieved performance, but has higher stability. There are no extreme values for start/stops in the table.

The results of the queue length-based autoscaler on series bender are not very good, probably because of its noisy nature. There would be no feasible parameters if the user needed to avoid ApdexC violations. The combination 2/0.5 results in significantly higher cost than the utilization-based case (972 vs. 791 h). The parameters 300/160 ms of the latency-based autoscaler, which produce similar performance, result in only 694 h. Even the start/stops count is higher. In most parts of the table, the start/stop counts are low, resulting in fewer changes performed and high cost. Parts of the table with down thresholds above 0.5 were trimmed because of excessive ApdexC violations.

The last examined series, gaff, when used with the queue length-based autoscaler, again shows the tendency of the algorithm to be unstable with low up threshold setting, similar to the latency-based autoscaler on the same series. While the number of machine-hours is

similar to the two other algorithms and the start/stop counts are lower by a factor of two, the ApdexC (hard SLO) violations are more frequent, also by a factor of two, making the algorithm unsuitable.

4.3.5 Second threshold for overload detection

However, while examining the results on virtually all three versions of the threshold-based autoscaler on the series gaff, we noted that the abrupt changes in level lead to extremely large queue lengths. The addition of only one machine still does not stop it from increasing. Actually, it is this series that forced us to implement the logic that estimates queue length in overload situations spanning multiple time intervals.

It is only logical to try to add more machines at once to prevent this from happening. We added a second, fixed, up threshold that adds four machines when the queue length is longer than 1000. Similarly, we have modified the latency-based algorithm with threshold 35s and the utilization-based one with threshold 90%.

Table 4.15: Queue length-based autoscalers on "gaff"

up	down	hours	starts	stops	ApdexC	ApdexS
2	0.13	*1595*	89	89	*32*	**3.2**
2	0.25	*1030*	*103*	*103*	*40*	5.43
2	0.38	799	*106*	*106*	*42*	6.02
2	0.5	680	*113*	*113*	*42*	8.92
4	0.13	764	**44**	**44**	*40*	5.2
4	0.25	**594**	**47**	**47**	*43*	6.32
4	0.38	**497**	**52**	**52**	*46*	9.14
4	0.5	**459**	**55**	**55**	*47*	*10.93*
6	0.13	723	**43**	**43**	*41*	5.58
6	0.25	**570**	**46**	**46**	*44*	6.62
6	0.38	**489**	**51**	**51**	*48*	9.52
6	0.5	**452**	**54**	**54**	*50*	*11.45*

Table 4.16: Queue length-based autoscalers with overload detection on "gaff"

up	down	hours	starts	stops	ApdexC	ApdexS
2	0.13	*1531*	92	92	11	**1.86**
2	0.25	*1013*	*108*	*108*	16	**3.35**
2	0.38	*807*	*114*	*114*	17	**4.24**
2	0.5	677	*120*	*120*	17	7.06
4	0.13	*937*	**56**	**56**	14	**2.9**
4	**0.25**	687	**63**	**63**	16	**3.94**
4	0.38	**574**	**70**	**70**	18	6.25
4	0.5	**526**	76	76	19	8.4
6	0.13	*937*	**56**	**56**	14	**2.9**
6	0.25	700	**65**	**65**	17	**3.94**
6	0.38	**588**	**74**	**74**	19	6.17
6	*0.5*	**538**	81	81	22	8.55

The result for series gaff was promising, as it significantly reduced the numbers of ApdexC violations and also improved ApdexS. The downside is a slight increase of machine starts and consumed machine-hours. The change made the algorithm competitive with the utilization-based one, although it is still worse. The settings 4/0.25 and 6/0.5 have similar metrics to the respective settings of the utilization-based autoscaler, but slightly higher cost: 687 and 538 vs. 618 and 491 h. The cost of the latency-based algorithm for the higher performing variant was only 556 h.

The effect of the additional threshold on other algorithms and series was mostly negative. It increases the number of machine starts and the cost with minimal impact on the grade of service, at least in areas of suitable configurations. There was no case, where this

modification would enable another solution or significantly improve the current ones. The only exception was again the series gaff and the latency-based algorithm. The effect there was a decrease in ApdexC violations across the table, similar to the queue length-based case. It enabled the solution 70/36 ms with 542 h, 91 starts, 17 ApdexC and just under 5% ApdexS violations. We infer that is only beneficial for series with abrupt changes in level and does not work well in the utilization-based case.

To conclude our experiment with the threshold-based autoscaler working on the queue length metric, we find it more stable (having less machine starts) than the other algorithms, which results in less hard and soft SLO violations, but at the cost of more consumed machine-hours. Also, similar to utilization, queue length serves as an early warning before a serious overload. This was best seen on series oe when contrasted with the latency-based algorithm, and on the experiment with overload detection on the series gaff.

The function of the up thresholds of the latency and queue length-based autoscalers are comparable, as there is a simple dependency between the two quantities, at least for Markovian service times:

$$R = \frac{(Q+1)S)}{N} \tag{4.6}$$

$$Q > N \tag{4.7}$$

where R is the response time, Q the queue length, SD the service demand, and N the number of servers. Nevertheless, specifying the down threshold in terms of queue length eliminates the biggest shortcoming of the latency-based autoscaler, that is the need to exactly specify the service demand of the application. The queue length-based autoscaler is remarkably insensitive to the down threshold setting. Any value between 0 and 1 basically means that the server is not overloaded. For an M/M/1 queueing system, it is precisely equal to the single server's utilization. A wrong setting will increase the cost and slightly decrease the user experience, but will not result in hard SLO violations.

4.3.6 The latency-queue hybrid autoscaler

We think that scaling up based on latency and down based on queue length should result in a low cost and high stability autoscaler. To support these claims, we have performed two more simulations of hybrid threshold-based autoscalers. The first proposed algorithm scales up based on latency and down based on queue length, with queue length used for level change detection on the series gaff, the second one scales up based on latency and down based on utilization, with queue length for level change detection on the series gaff.

The results table of the first proposed algorithm on the series oe seems to confirm the hypothesis. The algorithm takes the good properties from the latency and queue length-based algorithms, that is low cost and stability, respectively. The table has a single clear trend in both cost and performance metrics, similar to the utilization-based version, but has a wider band of feasible parameters, due to slightly lower achieved cost for the same performance. Concretely, the settings 60/1 and 70/1 result in 853 and 813 h, which is

Table 4.17: Latency-Queue hybrid autoscalers on "oe"

up	down	hours	starts	stops	ApdexC	ApdexS
50	0.25	997	**49**	**44**	**0**	**0.52**
50	0.5	947	**53**	**48**	**0**	**0.67**
50	0.75	923	**70**	**65**	**0**	**1.71**
50	1	907	81	79	**2**	**2.38**
60	0.25	937	**42**	**38**	**0**	**3.2**
60	0.5	**891**	**44**	**40**	**0**	**3.27**
60	0.75	**867**	**58**	**54**	**0**	**4.31**
60	**1**	**853**	**66**	**65**	**0**	**4.76**
70	0.25	**889**	**39**	**35**	**1**	6.54
70	0.5	**850**	**41**	**37**	**1**	6.77
70	0.75	**824**	**50**	**46**	**1**	7.81
70	*1*	**813**	**57**	**56**	**1**	8.4

Table 4.18: Latency-Queue hybrid autoscalers on "bender"

up	down	hours	starts	stops	ApdexC	ApdexS
300	0.13	*1112*	**3**	**0**	**1**	**0.3**
300	0.25	*1107*	**4**	**1**	**1**	**0.37**
300	**0.38**	893	**14**	**13**	**5**	**1.04**
300	0.5	792	**45**	**44**	19	**3.27**
500	0.13	*1080*	**3**	**0**	**5**	**0.97**
500	0.25	*1029*	**4**	**1**	6	**1.64**
500	0.38	826	**13**	**12**	16	**3.12**
500	0.5	**713**	**33**	**32**	*47*	9
700	0.13	*1029*	**3**	**0**	12	**2.38**
700	0.25	978	**4**	**1**	13	**3.05**
700	*0.38*	779	**11**	**10**	*26*	5.13
700	0.5	**672**	**28**	**27**	*74*	*12.49*
900	0.13	*1022*	**3**	**0**	13	**3.05**
900	0.25	971	**4**	**1**	14	**3.72**
900	0.38	770	**11**	**10**	*33*	6.32
900	0.5	**660**	**28**	**27**	*89*	*14.28*

lower than 918 and 844 h in the reference algorithm. The latency-based version is still better with 807 and 773 h.

Similar results were obtained for the series bender with the first hybrid algorithm. The chosen settings of 300/0.375 and 700/0.375 exactly match the selected settings of the utilization-based algorithm in the ApdexC metric and have lower cost in machine-hours: 892 and 779 h vs. 968 and 791 h. The latency-based algorithm was again better with 773 and 697 h.

The results of the first hybrid algorithm on series gaff only support the previous conclusion that it performs slightly better than the utilization-based version. The results for settings 70/0.375 and 130/1 are 617 and 510 h, which is not significantly worse than 618 and 491 h. The latency-based autoscaler with overload detection achieved 542 h in the first case (ApdexS violations below 5%). This hybrid algorithm was also better than the clean queue length-based variant on all tested time-series and fulfilled the expectation of lower sensitivity to the down threshold setting compared to the latency-based algorithm.

4.3.7 The latency-utilization hybrid autoscaler

The last model was created to test the hypothesis that queue length (when < 1) has similar properties as utilization. It uses latency for scaling up and utilization for scaling down.

As seen in the result table for series oe, this is not entirely true. While queue length provides high stability, utilization values over 50 resulted in oscillations and were trimmed

Table 4.19: Latency-Utilization hybrid autoscalers on "oe"

up	down	hours	starts	stops	ApdexC	ApdexS
45	10	*1013*	**52**	**47**	**0**	**0.22**
45	20	962	**65**	**62**	**0**	**0.37**
45	30	938	81	80	**0**	**1.26**
45	40	910	*103*	*102*	**2**	**2.45**
45	50	**890**	*136*	*135*	**3**	**3.72**
55	10	937	**45**	**41**	**0**	**1.86**
55	20	**886**	**49**	**46**	**0**	**2.01**
55	30	**869**	**62**	**61**	**0**	**2.68**
55	**40**	**844**	76	75	**1**	**3.72**
55	50	**825**	*104*	*103*	12	5.58
65	10	**890**	**40**	**36**	**1**	5.06
65	20	**843**	**43**	**40**	**1**	5.28
65	30	**823**	**51**	**50**	**1**	6.02
65	40	**804**	**64**	**63**	**2**	6.99
65	50	**790**	85	84	12	8.62
75	10	**866**	**39**	**35**	**1**	7.88
75	20	**821**	**41**	**38**	**1**	8.1
75	30	**802**	**46**	**45**	**1**	9
75	*40*	**789**	**57**	**56**	**2**	9.67
75	50	**768**	**73**	**72**	11	*11.3*

Table 4.20: Latency-Utilization hybrid autoscalers on "bender"

up	down	hours	starts	stops	ApdexC	ApdexS
200	10	*1055*	**6**	**4**	**0**	**0**
200	**20**	843	**58**	**57**	**5**	**0.74**
200	30	781	*148*	*147*	*30*	**4.24**
300	10	929	**2**	**0**	**0**	**0.15**
300	20	**727**	**35**	**34**	10	**2.23**
300	30	**680**	*109*	*108*	*44*	7.66
400	10	929	**2**	**0**	**1**	**0.22**
400	*20*	**693**	**27**	**26**	22	**3.79**
400	30	**642**	87	86	*67*	*11.82*
500	10	928	**2**	**0**	**2**	**0.3**
500	20	**683**	**24**	**23**	*30*	5.06
500	30	**620**	77	76	*96*	*15.61*
600	10	928	**2**	**0**	**2**	**0.3**
600	20	**665**	**23**	**22**	*39*	7.06
600	30	**595**	**67**	**67**	*126*	*19.33*

from the published version of the table. After the range of the down parameter was thus adjusted, the table looks clean enough. Moreover, the results with settings 55/40 and 75/40 are slightly better than the previous algorithm, 844 and 789 h for the higher and lower performance goal, respectively.

Similar properties were observed with the second hybrid algorithm on the series bender. The result table before trimming was mostly in italics, because the algorithm was unstable for down thresholds above 30. Using settings 200/20 and 400/20 again yields lower cost than the previous algorithm, 842 and 693 h. The second value is on the same level as 697 h of the clean latency-based autoscaler.

The results of the second hybrid algorithm on series gaff were trimmed not because it would be unstable at high down thresholds, but simply because the table is very flat and uninteresting. On the other hand, that means the algorithm was insensitive to its settings, which was our goal. One cause might be that the overload detection based on queue length is active and is performing most of the scale-up operations. The character of the table is very similar to that of the utilization-based autoscaler on this series, even the results are practically identical, settings 130/10 and 130/50 lead to 622 and 489 h vs. 618 and 491 h. Against the previous version, the results are also very close, so the conclusion is that with

Table 4.21: Latency-Queue hybrid autoscalers with overload detection on "gaff"

up	down	hours	starts	stops	ApdexC	ApdexS
70	0.13	*1008*	**63**	**63**	13	**2.01**
70	0.25	731	76	76	16	**2.9**
70	**0.38**	617	89	89	17	**4.09**
70	0.5	**559**	95	95	17	5.8
70	0.63	**547**	96	96	17	6.02
70	0.75	**545**	99	99	17	6.02
70	0.88	**542**	99	99	17	6.02
70	1	**545**	*102*	*102*	18	6.1
130	0.13	*964*	**58**	**58**	14	**2.83**
130	0.25	705	**67**	**67**	17	**3.94**
130	0.38	**588**	**73**	**73**	18	5.95
130	0.5	**533**	78	78	18	8.18
130	0.63	**520**	79	79	18	8.77
130	0.75	**516**	80	80	18	8.85
130	0.88	**510**	80	80	18	9
130	*1*	**510**	81	81	18	8.77

Table 4.22: Latency-Utilization hybrid autoscalers with overload detection on "gaff"

up	down	hours	starts	stops	ApdexC	ApdexS
70	10	642	78	78	15	**2.9**
70	20	**545**	95	95	17	5.58
70	30	**542**	98	98	17	5.95
70	40	**539**	*102*	*102*	18	6.17
70	50	**525**	*105*	*105*	19	6.54
130	**10**	623	**71**	**71**	18	**4.09**
130	20	**518**	78	78	18	7.88
130	30	**513**	81	81	18	8.77
130	40	**508**	81	81	18	8.77
130	*50*	**490**	81	81	18	9.22

this series, these three algorithms perform the same.

Even considering the fact that down thresholds higher than 50 are unusable, the stability of the latency-utilization hybrid is still better than that of the clean latency-based algorithm, and the cost per performance is better than the reference utilization-based algorithm and the previous hybrid one, making it the best algorithm in this study. It is also more feasible than the latency-queue length autoscaler from an implementation perspective. It is much more commonplace to monitor the utilization of servers and latency of applications than to monitor average queue lengths on load balancers.

4.3.8 Summary of results

In this subsection, we present Table 4.23, which summarizes the results of the different autoscaler algorithms working on our three sample time series. The chosen settings were already presented in bold and italic on the left side of each table and represent the lowest cost in machine hours attainable using the particular algorithm, which had soft SLO violations in less than a) 5% and b) 10% of the 15-minute intervals. Only in the case of bender, the criterion for selection was the ApdexC, or hard SLO violations, as explained in the text.

Table 4.23: Summary table

series	alg	up	down	hours	starts	stops	ApdexC	ApdexS
oe	utilization	**70**	**20**	918	**51**	**48**	**0**	**4.83**
oe	latency	**60**	**34**	**807**	**66**	**65**	**0**	**4.83**
oe	queue	**N/A**	**N/A**					
oe	lat-que	**60**	**1**	**853**	**66**	**65**	**0**	**4.76**
oe	lat-util	**55**	**40**	**844**	76	75	**1**	**3.72**
oe	utilization	*80*	*20*	**844**	**46**	**43**	**1**	9.07
oe	latency	*70*	*34*	**773**	**50**	**49**	**2**	8.33
oe	queue	*4*	*0.63*	913	**53**	**46**	**0**	9.52
oe	lat-que	*70*	*1*	**813**	**57**	**56**	**1**	8.4
oe	lat-util	*75*	*40*	**789**	**57**	**56**	**2**	9.67
bender	utilization	**80**	**10**	968	**7**	**6**	**5**	**1.19**
bender	latency	**300**	**152**	773	**15**	**14**	**5**	**1.19**
bender	queue	**N/A**	**N/A**					
bender	lat-que	**300**	**0.38**	893	**14**	**13**	**5**	**1.04**
bender	lat-util	**200**	**20**	843	**58**	**57**	**5**	**0.74**
bender	utilization	*70*	*20*	791	**34**	**33**	*26*	**4.83**
bender	latency	*700*	*152*	**697**	**5**	**4**	15	**3.79**
bender	queue	*2*	*0.5*	972	**55**	**53**	*31*	5.13
bender	lat-que	*700*	*0.38*	779	**11**	**10**	*26*	5.13
bender	lat-util	*400*	*20*	**693**	**27**	**26**	22	**3.79**
gaff	utilization	**70**	**10**	618	**67**	**67**	16	**3.87**
gaff	latency	**50**	**36**	**556**	100	100	19	**3.2**
gaff	latency-ol	**70**	**36**	542	91	91	17	**4.98**
gaff	queue-ol	**4**	**0.25**	687	**63**	**63**	16	**3.94**
gaff	lat-que-ol	**70**	**0.38**	617	89	89	17	**4.09**
gaff	lat-util-ol	**130**	**10**	623	**71**	**71**	18	**4.09**
gaff	utilization	*70*	*50*	**491**	80	80	17	9.07
gaff	latency	*N/A*	*N/A*					
gaff	latency-ol	*N/A*	*N/A*					
gaff	queue-ol	*6*	*0.5*	**538**	81	81	22	8.55
gaff	lat-que-ol	*130*	*1*	**510**	81	81	18	8.77
gaff	lat-util-ol	*130*	*50*	**490**	81	81	18	9.22

4.3.9 Stability of the hybrid algorithms with increased load

New autoscaling methods should be investigated, because, as we have theoretically shown, the most commonplace utilization-based autoscaler will become unusable with machine counts over about 32, when the utilization-latency plot of the M/M/N model becomes very close to the axes and finding a right setting for the up threshold in the utilization

metric will become hard.

We have simulated this in our model by multiplying the first tested time series with powers of 2. The multiplier used in the study was 16. When set to 8, the optimal up threshold to get the ApdexS under 75% violations below 5% of the time was 60%. At 16, it was 70%, at 64, it was 80%, at 128, it was 90%, and at 256, the optimal value was not present in the table. We would have to search for it between 90 and 100%. Lower up thresholds of course work, but result in wasted processing power and money. At the last traffic level and settings 90/50, the 95% percentile of machines used was 63.

When the same experiment was repeated with the proposed hybrid threshold-based autoscalers, it was noted that the average queue length is a function of the load (as the mean arrival rate increases, so do the "spikes", or better said the variance of the Poisson distribution that governs the arrival rate). This caused the latency-queue autoscaler to quickly stop scaling down correctly as the incoming traffic was increased. The down threshold had to be increased up to about five at the highest traffic setting.

On the other hand, the latency-utilization autoscaler worked with the same settings for all tested traffic levels, which confirms that the latency, as a function of the application and SLO, and utilization, at light loads only weakly dependent on the number of nodes, are the best choices for thresholding metrics at any incoming traffic level.

4.3.10 Simulation plots

In the section, we present the plots that are the output of the simulator when used interactively. They show the user the input time series along with all computed variables: utilization, queue length, response time, the number of machines used, and the simulated user satisfaction expressed using Apdex.

The plot 4.13 is of the time series oe and the latency-queue hybrid autoscaler with settings 60 ms up and 1 req down. It presents a good compromise between cost and performance as it keeps the response time very low, but not at the absolute minimum. The only downside is oscillations that happen during periods of low load.

The next two plots show the time series bender and the classic latency-based autoscaler with settings 300 ms up and 152 ms down. This time series has a much lower seasonal component and also lower request intensity, so there are fewer scaling actions. However, the more conservative autoscaler results in gradual increases in response time, which falls abruptly after each scaling action. Sometimes, the action comes too late, and the growth in latency is felt by the customers. The first plot shows another output mode of the application, while the second shows a diagnostic printout of the different Apdex components. As a reminder, ApdexS has thresholds T=200 ms and F=1 s, while ApdexC is set up with T=1 s and F=4 s.

The plot 4.16 represents the time series gaff and the latency-utilization hybrid autoscaler with parameters 130 ms up and 10% down. The overload detection is on and reacts to queue lengths above 1000 req by adding four instances at once. This load profile is impossible to scale without overloads using a reactive autoscaler, but the long queue detection at least reduces the time the system is overloaded.

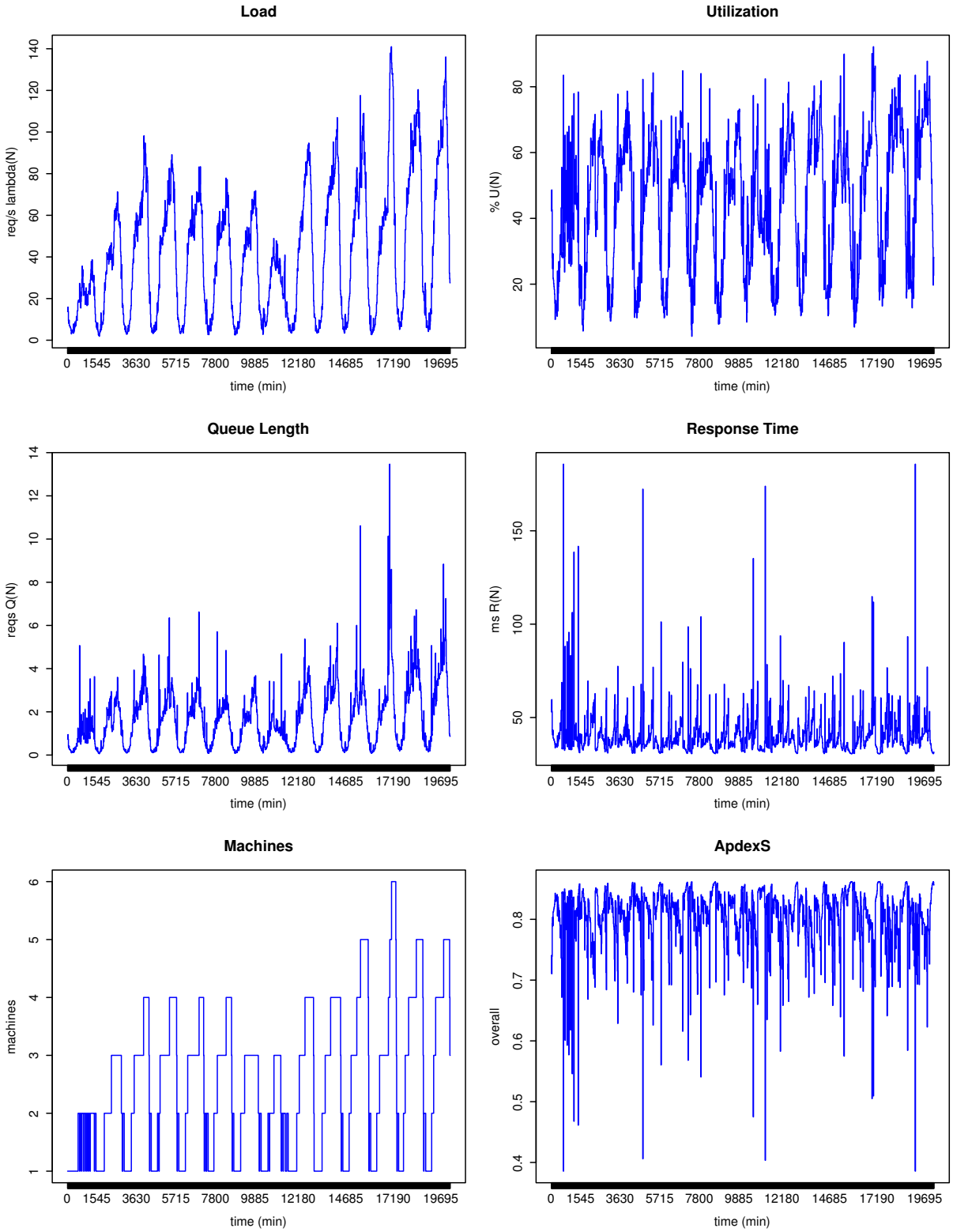


Figure 4.13: Time series oe and the latency-queue hybrid autoscaler with settings 60 ms up and 1 req down. Six figure plot mode.

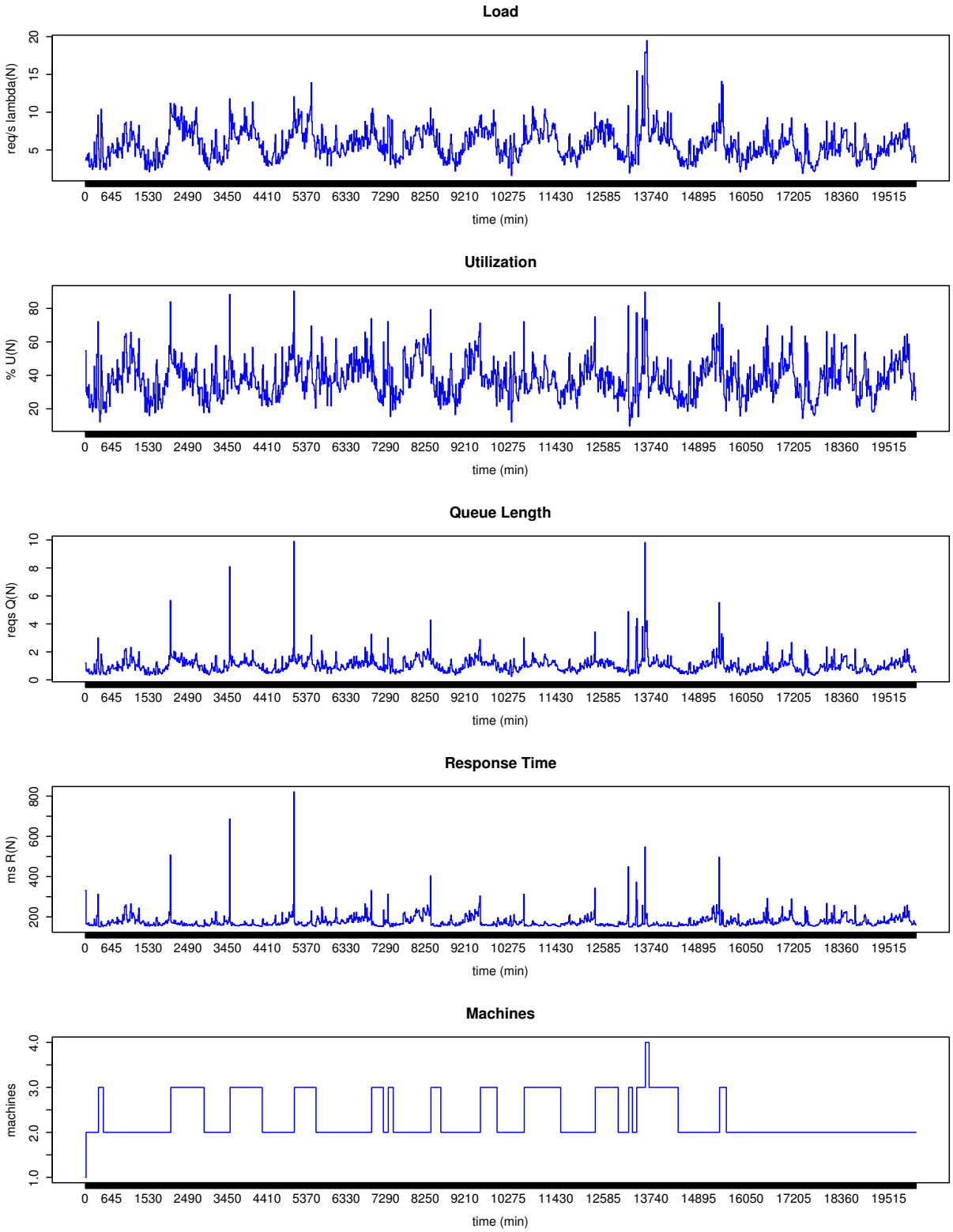


Figure 4.14: Time series bender and the classic latency-based autoscaler with settings 300 ms up and 152 ms down. Five figure plot mode.

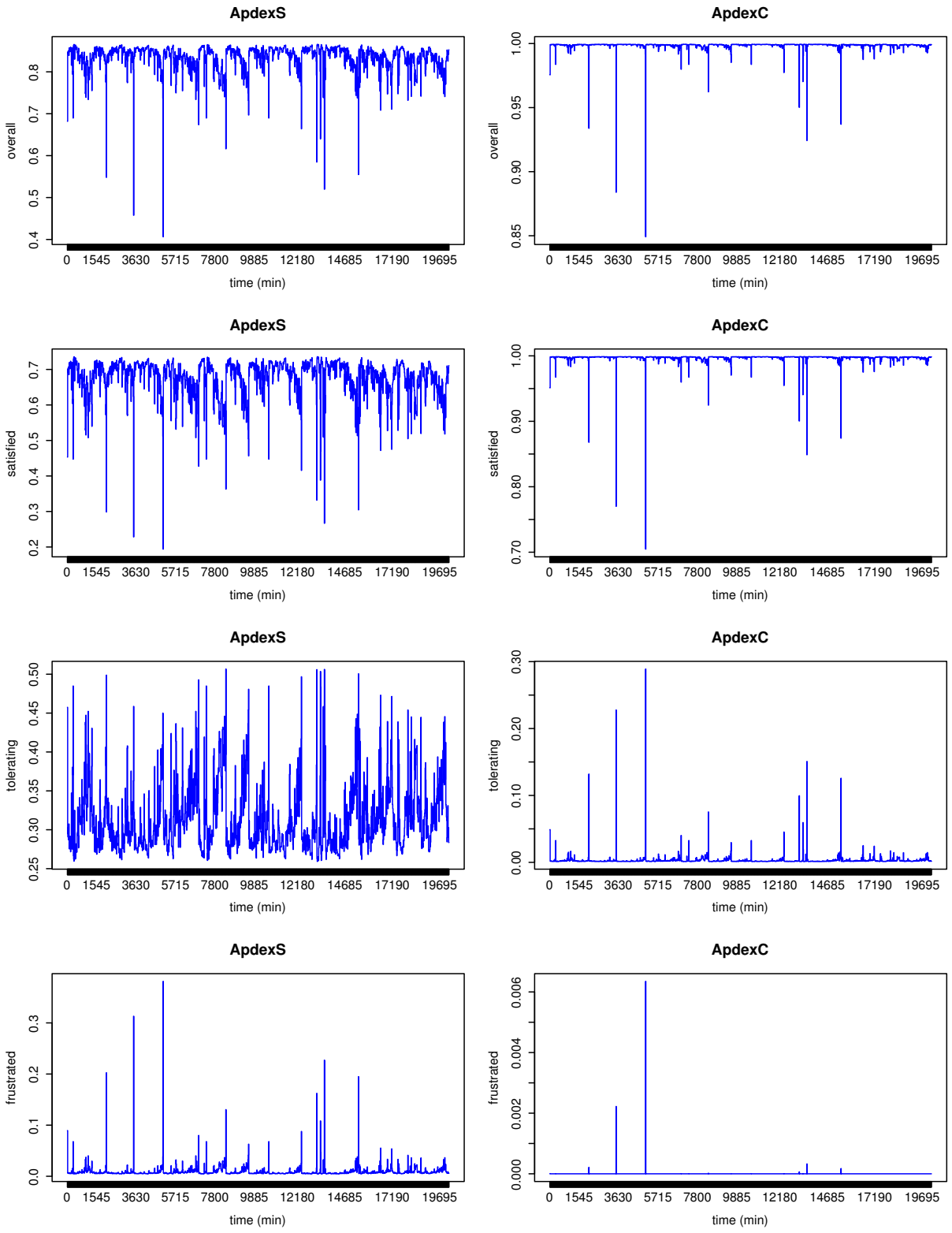


Figure 4.15: Time series bender and the classic latency-based autoscaler with settings 300 ms up and 152 ms down. Apdex component plot.

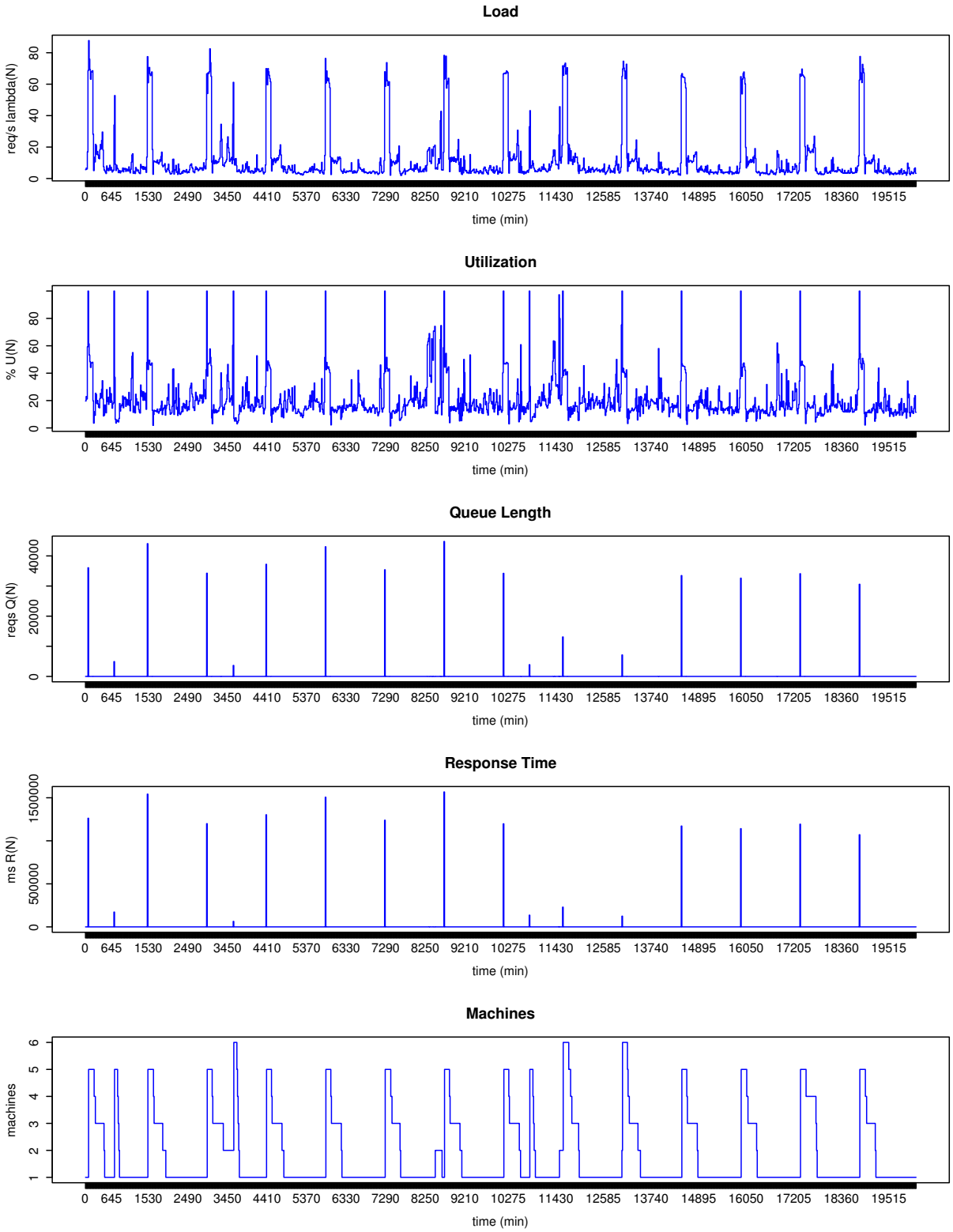


Figure 4.16: Time series gaff and the latency-utilization hybrid autoscaler with parameters 130 ms up and 10% down with overload detection. Five figure plot mode.

4.4 Minor results and developed software

Besides the main results detailed in this chapter, several analyses and software packages have been developed around the main line of work, mainly as master's and bachelor's theses where the author of this work posing as either the supervisor or a consultant (mentioned in Acknowledgements). They will be presented briefly in this section.

4.4.1 Cloud Gunther

The Cloud Gunther is a batch job queuing engine that has integrated cloud instance management. It was first presented at the POSTER conference [88] and later, along with the research idea of using heterogeneous load to maximize cloud utilization at the IARIA CLOUD COMPUTING conference [89].

The program has been designed to make use of spare computing power of lab workstations for scientific computations. There are several tools that can be utilized for that purpose, but we have chosen to use cloud technology because it allows running a broad spectrum of computations utilizing different languages and libraries. At the same time, it does not require any further modifications to the lab workstations bar the installation of the cloud software, as all computations are run inside virtual machines whose templates can be prepared by the users. This is an improvement over grids and clusters, where every software package has to be installed by the administrator directly on the nodes.

The program itself is written in Ruby on Rails and consists of a web application for job submission and monitoring, several daemons that handle job scheduling and output collection and a messaging service to deliver jobs to instances. The design is detailed in the master's thesis of Josef Šín [90].

The queuing logic is rather simple and is outperformed by traditional cluster computing tools. The differentiating feature of the program is its ability to provision cloud instances just before they are needed and to delete them afterwards. More specifically, it launches a specified number of instances from a certain template for a certain user and will use them to run tasks as long as there are suitable jobs in the queue. It can also monitor the remaining amount of resources in a private cloud and only launch jobs if there are enough VM slots left. The system also has the property of multitenancy, meaning that it can serve multiple users without mixing their jobs and virtual machines. Different job priorities for different users are also implemented.

The system was developed on the now outdated Eucalyptus v2 private cloud, which ran on about 20 workstations. This cloud implements the Amazon Web Services API, so Cloud Gunther can also be used in the public cloud or, with slight modifications, for other private clouds. They differ mainly in the function for presenting the available capacity, which is not a part of the (public cloud) standard. The component for reading available capacity directly from the cloud could also be easily replaced by a reading a prediction of free resources to accommodate other load on the cluster and leave room for its fluctuations.

Cloud Gunther was tested in cooperation with other researchers. The problems solved were an image transformation that ran on tens of gigabytes of data in a pseudoparallel

fashion and was written in Matlab. The second experiment was with a multiagent simulation that required the nodes to communicate with each other. It was written in Java. A robotics simulation was not performed in the end because it required computation-capable graphic cards and there is no way for them to be shared between virtual machines. They can be connected only to one at a time, and even that function is, as far as we know, absent from private cloud software.

The project is currently dormant because there was no further demand for it and consequently, there was no will to maintain the cloud installation on the lab workstations.

4.4.2 ScaleGuru

ScaleGuru is an automatic scaling subsystem for private clouds modelled after Amazon Web Services' autoscaler. It has been published in the International Journal of Systems and Measurements [3] and details are also available in Karol Danko's thesis [91].

As already mentioned, private clouds are useful for experiments as those do not cost money and can be much better reproducible than in the shared environment of a public cloud. The problem is that they do not yet offer the same level of services. Particularly, automatic scaling is necessary if data about variable load is to be collected for forecasting experiments. At the time of writing of the software, there were autoscalers being written for at least Eucalyptus and OpenStack but were not yet finished. Creating an in-house autoscaler had the added benefit of greater simplicity of expansion.

The program can be deployed either by the cloud administrator on a controller machine or even by the client himself in a virtual machine. It is a web application written in Node.JS with the MongoDB database for settings and performance data. It utilizes Nginx as the load balancer and automatically creates configuration files for it on every topology change. It can manage multiple applications at once. Instance monitoring is done through injecting a small script that periodically sends low-level system statistics to a web application endpoint. The cloud is controlled through a wrapper to euca2ools, a package of Amazon Web Services-compatible command-line utilities from Eucalyptus. All these components are modular and can be extended if necessary.

The autoscaling logic itself is defined in terms of Autoscaling Groups, which define applications and their usage limits, Load Balancers, which describe the ports and domain names of applications, and in Autoscaling Policies, which react to Autoscaling Alarms set on monitoring data. For example, an alarm could fire, if the CPU load average of the instances serving an application was over 70% for 5 minutes. A policy for this alarm would then add a predefined number or a percentage of instances.

This scheme has the benefit of being well known and tried from the public cloud implementation at Amazon, and thus enables a seamless transition from public to private cloud. Using ScaleGuru for experiments has the benefit of knowing its code and being able to modify it if necessary. Also, all performance data is saved in the database for later analysis.

The program was so far only evaluated in a synthetic setup consisting of a load generator, the ScaleGuru VM also hosting the load balancer and a Eucalyptus v2 private cloud

of 8 nodes. The managed applications were a static file, a synthetic CPU bound task and a bare installation of Wordpress. In the first case, the load generator overloaded first, after creating about 10000 simultaneous users. The second variant was used more during the development of the monitoring and scaling components, and the web application was then used to verify the functionality.

During the experiments, the autoscaler was capable of scaling the application from one to eight virtual machines and back, the load varied from 1 to 50 simulated users, each issuing a request about 2x per second. The load balancer was only lightly loaded, but higher traffic could not be verified because of the small scale of the private cloud. The base latency was about 100 ms, and it remained below 2 seconds. If less was desired, the up-scaling threshold could be set lower to compensate for the time it takes to launch a new virtual machine, but that would lead to more wasted resources in steady load states.

4.4.3 Private IaaS cloud comparison

Traditional virtualization tools employed in large data centers are competing against increasingly popular cloud counterparts. The bachelor's thesis of Martin Klepac [5] focuses on a subset of cloud systems – private IaaS clouds, which allow running virtual machines at a large scale. As the overall private IaaS industry is fast-growing, new versions of the established systems arise very quickly, and therefore it is easy to lose track of the supported features. Additionally, some large IT players are waiting in the background to promote their favorite product without hesitating to influence the media attention.

Hence, to provide an unbiased overview, the thesis has performed a comparison of private IaaS cloud systems based on a range of supported features, installation and subsequent configuration complexity, quality of provided documentation and last but not least, on a quantitative measurement of virtual machines boot up time. The result of this is a list of fields, in which respective systems excel and on the other hand, fields in which they do not particularly succeed.

The results are very relevant to the process of selecting a private cloud implementation and as such the thesis has won a prize from the Czech ICT Alliance.

4.4.4 Private PaaS cloud comparison

Likewise, the master thesis of Pavel Pulec [92] attempts a study of available platform clouds. PaaS is a very young category of the cloud which allows developers to concentrate just on the implementation of web applications and does not bother them with the preparation of the platform. The goal of the thesis was a comparison of current PaaS clouds with emphasis on their advantages and disadvantages. A part of the thesis is also a demonstration of overload and analysis of behavior during the scaling of applications, which should be a part of every cloud platform.

During this work, we have discovered that the selection of PaaS systems is very limited and that their maturity considering mainly documentation, ease of deployment and opera-

tion, and functions, is severely lacking. Nevertheless, the author of this dissertation thesis is currently working on implementing a public installation of the Cloud Foundry PaaS.

4.4.5 Scheduling algorithms in job queues

The master's thesis of Jakub Tkadleček [93] addresses the problem of scheduling jobs in queues, focusing on the fairness of the planning. The work includes efforts to define a common structure of jobs arriving at the scheduler and evaluation of the impact of this structure on the planning process.

This work was an attempt to extend the Cloud Gunther job scheduler with more complex scheduling algorithms. The results are mainly a theoretical research of existing job scheduling algorithms and an analysis of data from the VIC ČVUT - a supercomputing center supporting the CTU in Prague. A simulator of several algorithms was written, and the data was fed to it. The result was that the users' estimates of job run time are so inaccurate that actual scheduling algorithms will not perform better that a simple FIFO queue with backfilling.

4.4.6 Data movement in Hybrid Clouds

The main focus of the bachelor's thesis of Matej Uhrín [94] is on hybrid clouds and web applications running in hybrid cloud mode. The thesis provides a method to decide whether it is more economical to run an application in multiple locations in hybrid cloud mode with all database accesses going through an Internet line or to replicate the database between the locations.

This thesis further species metrics and variables used in the decision method. Moreover, a profiling process on how to perform measurements and collect the mentioned variables is explained. The work finishes with the method being tested on two of the most common web applications.

The work focused on the behavior of web applications with regard to databases and the impact of the database accesses on the operating cost of the application in a public cloud. We have laid a methodology to measure all necessary variables and to let the deployer of the application solve the trade-off between the cost of the public cloud infrastructure and the quality of service for the end users.

4.4.7 Automatic cloud deployment driven by a performance model

The goal of the master's thesis of Tomáš Kábrt [95] was to analyze the current state of IaaS (Infrastructure as a Service) cloud technologies and review the possibilities of automatic provisioning and deployment of a two-tier application which consists of phpMyAdmin and Percona Cluster with the Chef orchestration tool. This application was benchmarked, and a performance model was created. The ideal number of nodes for the application was calculated by PDQ based on the performance model. The ADT web application was created to offer an easy way to deploy nodes with required software in the cloud automatically.

The motivation behind the project originally called "The Magic Cloud Provisioner" was to be able to suggest to the user the scale of an open-source web application taken from a catalog of benchmarked applications given his estimated of the demand for his or her content and to deploy the application at the computed scale. This goal has been achieved using the DevOps tool Chef for application deployment, classical performance testing of the applications and then modeling them using the PDQ Queueing Network solver.

This model is perhaps the most important result because this algorithm would need to be a part of a predictive autoscaler that would predict the user demand for the application and not the utilization directly. This piece would then compute the desired scale of each application tier and notify the autoscaler to add or remove nodes accordingly.

4.4.8 Private cloud monitoring

The goal of the bachelor's thesis of Matěj Židek [96] was to create an enhanced monitoring system for the cloud platform Open-Nebula that would store data for a duration of at least one year, and present is in graphs in a user-friendly environment.

The final solution uses the one2influx daemon to gather monitoring data from Open-Nebula's XML-RPC API and saves them to the InfluxDB database. The visualization is done using the Grafana tool, and after a customization of one2influx, the solution applies to other cloud solutions as well, e.g. OpenStack.

At the moment of the writing, the solution was in a state of limited functionality. The problem was caused by InfluxDB, which was in heavy development at the moment. This work thus serves more as a demonstration of what can be achieved with the use of InfluxDB in a short period.

As far as we know, the solution has been used at the Institute of Computer Science at Masaryk University in Brno at least as inspiration for their new solution for OpenNebula cloud monitoring. The institute runs the cloud for the benefit of all research organizations in the Czech Republic, and there is normally no metering and accounting in the private cloud software. They do not do billing in monetary units, so monitoring of fair usage is necessary to maintain the availability of the system for the community.

4.4.9 Autoscaling in Cloud Foundry

The progressive monitoring solution with the now mature InfluxDB time series database has also been used in the bachelor's thesis by Maroš Špak [97] that worked on the higher level of PaaS.

Cloud Foundry is an open-source cloud platform as a service. It aims to provide a simple and a quick way to deploy, maintain and scale applications in the cloud. Currently, Cloud Foundry supports only manual scaling of the applications via the command line tool or the HTTP API. It also lacks the storage system to permanently store the applications information about used system resources. That presents a problem for the automation of application scaling. The aim of the thesis was to analyze the possibility of the application

scaling on the Cloud Foundry platform and to design and implement the missing auto-scaling function including retrieving and storing data from the application's resource's usage monitoring.

The implemented program is written in the Python language. It monitors the application resources usage for each of the application's instances. The retrieved data is stored in the InfluxDB database. With the help of the user-defined scaling rules, this data is analyzed, and the program evaluates the need for scaling the application. An algorithm used to evaluate the collected data is implemented as a plug-in and can be replaced. The configuration of the scaling, which includes scaling rule definitions, is stored in the MongoDB database. For the purpose of changing these settings, the program provides a simple HTTP API, making it ideal for inclusion as a microservice into a bigger PaaS cloud management platform.

The scaling itself is provided by the Cloud Foundry via its HTTP API, and it is handled by changing the number of running instances, the amount of allocated memory, or disk space.

Chapter 5

Conclusions

This dissertation thesis presented the results of our work focused on increasing the utilization of private clouds. In particular, we have discussed the benefits of automatic scaling for the cloud user and the impact it will have on the provider's side. We have focused on optimizing the client side of IaaS because of popular demand and the poor availability of data center scale data.

5.1 Workload Forecasting

The first part of the work presented two methods of time series forecasting, used otherwise mainly in economic forecasts, and which could be applied to server load data. These methods were tested on six time series of CPU load, some of which are web servers with a well defined daily curve (oe, bender, wn), and some have a load of more unpredictable nature (lm, real, gaff).

The results of the forecasts are very promising. We believe that in practice, we should mainly encounter series that are not difficult to forecast because computer systems with human interactive use will always exhibit strong seasonality, which makes them easily predictable. Secondly, those web services which actually need autoscaling will have enough traffic for the time series to be smooth, without spikes due to random arrivals of individual requests. Lastly, there is a trend to separate the interactive frontend and its backend services, so time series like gaff where user accesses and database imports ran on the same virtual machine will be less and less frequent. The series which did not have good properties were included in the study mostly to test the limits of the forecasting methods.

Our proposal for optimizing the utilization in private clouds is to periodically evaluate the long-range workload forecast and use the result to either shut down idle machines when there will be no demand for them for longer than a specified threshold (to minimize stress on the computers due to frequent on/off cycles), or to query the result of the forecast algorithm in a batch job scheduler (including the now modern big data platforms), and thus ensure that the unused capacity will be filled with non-interactive traffic.

For this proposal to be viable, the cloud should contain load-balanced and autoscaled web servers as the variable component that is to be predicted. However, during the study, we have not found a private cloud installation that would employ cloud elasticity. We have therefore shifted focus to cloud autoscaling simulations to prove the advantages of autoscaling and show application administrators that the trade-off between cost and customer satisfaction that exists in autoscaling and is probably the thing they are most afraid of, can be controlled in advance by choosing the right parameters.

5.2 CloudSim Changes

In order to be able to evaluate autoscaling algorithms, the CloudSim simulator package was studied and modified by adding several functions necessary for the testing of cloud automation software, most notably VM addition and deletion at run time and more detailed statistics collection. The modifications allowed a simple autoscaler to be implemented.

Due to concerns about the accuracy of the simulation, a load test of a dynamically changing virtual infrastructure was set up and compared with the simulator. The results were significantly different from the measurements on the real system.

While the simulator contains the possibility of generating traffic and thus conducting simulated load tests, its focus is not the workload itself, but rather the testing of cloud systems with different characteristics. This is also backed by the focus of other articles using CloudSim for evaluation [33][34][35], which focus either on Green Computing or on simulation batch workloads. CloudSim is arguably better suited for those than for large numbers of small requests, because of its origin in GridSim.

The search for a more accurate simulator turned empty. Other simulators than CloudSim and its derivatives, which are targeted specifically at cloud computing, are in most cases specialized to infrastructure simulations and cannot simulate the kind of scenarios we need. The only two exceptions found are CDOSim [39], a part of CloudMIG Xpress. The article also contains the only other known comparison of CloudSim-based software with a real load test. The software is, sadly, not available any longer. The second one is the SimuLizar plugin [51] for PCM [48], whose evaluation in the article does not seem to show all its possibilities. Other articles are mostly working on datacenter scale, where the cost of the real test would be overwhelming. Nevertheless, we think that every simulation should be verifiable, at least on a small scale.

With no viable alternative simulator, the problems with CloudSim have been resolved by backporting queueing logic from the latest version, fixing order errors this introduced due to different time units across CloudSim versions, mitigating rounding errors that caused incorrect results, implementing a FIFO queueing discipline for increased correctness and as a reference for the default PS discipline, fixing the load generation code, and including service time randomization to simulate a Poisson process of both arrivals and departures.

The modified simulator was verified both against the results of the original load test and against a queue theoretic model of the test setup. All three experiments give the same results (with error percentages in single figure numbers).

Therefore, the main contribution of this part of the work is the proof that the modified version of CloudSim is correct and can be used to simulate interactive traffic. The steps to add the necessary functions and the steps required to fix the queueing logic are documented. The modified CloudAnalyst code is available on GitHub[1].

For smaller scale experiments, no simulation is needed and live load testing can be performed using load generators and performance monitoring tools. As seen in Section 3.2.2, the setup of these experiments is not very demanding and the instrumentation can capture a wide variety of variables, perhaps more than most simulations. Cloudstone[2] from Cloudsuite [98] is a standardized benchmark targeted at web serving, which is very similar to the one presented here. It uses the PHP application Olio, a mock-up of a social site, instead of Wordpress. Working implementations of an autoscaling controller and a job queue for private clouds are already available at our university.

5.3 Custom simulator design

Due to the difficulties encountered with CloudSim, we have designed our own simulator for autoscaling scenarios. It is based on the PDQ queueing network solver [42] with some additions to compute queue lengths in overload situations. The designed queue-theoretic model-based simulation script in R passed our small-scale verification well, in fact even better than our best version of CloudSim. This thesis presented its design and possible variants, as well as showing its results on basic threshold-based algorithms.

The simulator takes a workload trace of request intensity and iterates over it, computing the queue length, latency and utilization of the service system. All these variables and their historical values are available to the autoscaling algorithm, which decides on the number of servers for the next iteration. This way, autoscalers reacting to latency (Google App Engine type), utilization (Amazon Web Services Type), or queue length (RedHat OpenShift type) can be simulated. With the time series analysis tools available in R, it is possible to test designs of proactive autoscalers based on some form of time series forecasting.

While evaluating our model on various threshold-based autoscaling algorithms using different metrics, we have found out that a combination of scaling up based on latency and down based on utilization is better than using any individual metric alone regarding both stability and cost per performance. The arguments are presented in Subsection 4.3.9.

As the used model is analytical, the resultant parameters also have known distributions. Using formulas for GoS on an M/M/N system and the Apdex (Application Performance Index), we can calculate the performance of the system as perceived by users. On the other hand, the model computes the cost for running the cloud system expressed in machine-hours. This way, the simulator helps the user choose a compromise between cost and performance. See Subsection 4.3.1 for explanations of the model parameters and outputs. If the user can quantify both the cost per machine-hour and the dissatisfaction of users

[1] `https://github.com/vondrt4/CloudAnalyst-interactive`

[2] `http://parsa.epfl.ch/cloudsuite/web.html`

with performance in the same units, e.g. money, then the model can give them a single output metric. This was not done in our evaluation.

While the evaluation was done on a single-tier autoscaled service, the benefit of using the QN formalism is that it supports simple extension to multiple tiers, just by adding more service centers to the model. The code is already available. This allows simulating autoscalers that see the utilization of all tiers at once. Otherwise, when a multi-tiered application has autoscalers between tiers, which are only concerned with their respective tier, one scaling action may only solve the worst bottleneck, and can move it to another tier. Monitoring all tiers allows scheduling complex scaling actions, as shown by Urgaonkar [99]. The QN solver also allows for multiple input streams with different service demands, which contribute to the overall utilization.

The drawbacks of the chosen model engine are that the MVA (Mean Value Analysis) class of algorithms available in PDQ simulates a network of M/M/N queues. Therefore, it cannot model highly bursty traffic, where arrivals are not Markovian. Other modeling packages may, however, be chosen, which support more arrival distributions. The overall design of the simulation script will not be changed. The list of possible choices, along with a selection of discrete event simulation engines, is presented in Subection 2.2.2.

Forecasting of computer load time series, as presented in the first part of the thesis, uses the effect of combined load of many requests to create a continuous curve of load. This works if the number of requests is sufficiently high not to see the individual requests. Continuity can be expected from measurements of an entire cloud. However, if an application that has a low number of users but a high service demand per request were to be autoscaled using prediction, the effect of the random arrival of requests would become visible. In these cases, direct load forecasting methods become unreliable, and usage of performance modeling to compute the load would be necessary for the project of predictive autoscaling. In these cases, the queueing network model should be embedded in the autoscaler. If the prediction is done on a trace of incoming requests and not on the load trace, as usual, these problems could be mitigated.

To prove an autoscaling algorithm using our simulator, the researcher will need traces of request intensity from a live cloud application as well as the baseline latencies, utilizations, and virtual machine numbers. Such traces are not publicly available. The traces upon which we tested the simulator itself also do not satisfy these conditions. They were taken from single-machine web servers of different purposes. The requirement of having request intensity traces was relaxed by using utilization traces and stating that the servers were running in low-load mode all the time. We have used current readings of latency to obtain the value of the service demand for the simulation.

We hope that the simulator will be used both in research to test new autoscaling methods and in practice for choosing the right method and parameters for a particular workload. The source code in R is available on GitHub.[1]

Combined with our work on load prediction, it should be possible to design and simulate

[1] `https://github.com/vondrt4/cloud-sim`

a pro-active autoscaler, which can anticipate both increases and decreases in application demand and act accordingly.

5.4 Summary

In agreement with the goals stated in the Introduction, we have studied the nature of interactive cloud workloads, mainly through the disciplines of Time Series Forecasting and Queueing Theory. This study led us, in the first part, to formulate our solution for private clouds, which is to employ autoscaling of the interactive part of the workload and forecast the resulting curve to be able to either save power or efficiently backfill the free capacity with batch jobs.

The dependency on the interactive workload relinquishing unused capacity led us to study further the properties of different autoscaling algorithms working on various metrics available in an autoscaled system. To do that, we created two simulation methods. One is based on the CloudSim event-driven simulation framework and can be modified to simulate different request arrival distributions or other non-standard conditions, while the second one based on queueing theory is much faster, although it is more constrained by assumptions about the properties of the input streams.

Our strategy for the public cloud or generally for reducing the operating cost of interactive workloads is based on this second simulator and entails measuring the computational complexity and the workload time series of the application, running a parameter sweep of different autoscaling algorithms and their parameters, and then choosing the one that makes the right compromise between cost and customer satisfaction according to the customer's service quality needs.

5.5 Contributions of the Dissertation Thesis

In particular, the main contributions of the dissertation thesis are as follows:

1. New methodology for forecasting time series of web server load and its validation
2. Extension of the often-used simulator CloudSim for interactive load and increasing the accuracy of its output
3. Design and implementation of a fast and accurate simulator of automatic scaling using queueing theory

5.6 Future Work

As future work, we would like to integrate the two parts of the thesis into one project which would enable the maximization of utilization of a private cloud. However, to do that, we would need some real test bed with user traffic. In particular, the forecasting

strategy currently requires some manual work before being deployed, mainly to select the right learning horizon and retraining interval, as these depend on the rate of change of the traffic. Therefore, it is best applied manually.

Given that the methodology for model parameter selection can rely on an algorithmic evaluation of statistical tests at every step, it is possible to implement an automatic version from which even layman users can benefit, for example in the form of a pro-active autoscaler. It will probably not be as accurate as when the manual approach is used. Retraining of the model would be done on manual request, or when a change in time series parameter would be detected or if the prediction accuracy were to fall.

We would like to explore other ways to employ predictions in autoscaling, concretely methods to predict spikes and other deviations from the daily curve at their early onset. Currently, predictive autoscalers need to be paired with reactive ones to cover the times when the load deviates from the forecast. The reaction of the autoscaler could be improved even in these situations using some form of anomaly detection.

Alternatively, we could apply our prediction techniques to other areas, such as data center cooling and power distribution, or use the queueing network formalism to describe big data analytic platforms and algorithms.

This dissertation thesis has laid the groundwork for optimization in both the private and the public cloud. What remains now is to put the recommendations into practice. The author is currently working on building a public cloud data center in the Czech Republic. Automatic scaling is being offered to any client with high enough resource demands. Once the scale of the data center gets high enough, optimization strategies will, of course, be built in.

The autoscaling simulator is a complete solution for algorithm evaluation for researchers or parameter assessment for users.

Bibliography

[1] Wickremasinghe, B. *CloudAnalyst: A CloudSim-based Tool for Modelling and Analysis of Large Scale Cloud Computing Environments*. CSSE Dept., University of Melbourne, 2009, mEDC Project Report. URL `http://www.cloudbus.org/students/MEDC_Project_Report_Bhathiya_318282.pdf`

[2] Calheiros, R. N.; Ranjan, R.; Beloglazov, A.; et al. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, volume 41, no. 1, 2011: pp. 23–50.

[3] Vondra, T.; Šedivý, J. Maximizing Utilization in Private IaaS Clouds with Heterogenous Load through Time Series Forecasting. *Int. J. On Advances in Syst. and Measurements*, volume 6, no. 1 and 2, 2013: pp. 149–165.

[4] Illich, M. Cloud je někdy také předražená hračka. June 2011, [Online; posted 22-June-2011]. URL `http://www.lupa.cz/clanky/cloud-je-predrazena-hracka/`

[5] Klepac, M. *Private IaaS cloud comparison*. Bachelors thesis, Czech Technical University, Faculty of Information Technology, 2013.

[6] Moore, L. R.; Bean, K.; Ellahi, T. Transforming reactive auto-scaling into proactive auto-scaling. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, ACM, 2013, pp. 7–12.

[7] Weingärtner, R.; Bräscher, G. B.; Westphall, C. B. Cloud resource management: A survey on forecasting and profiling models. *Journal of Network and Computer Applications*, volume 47, 2015: pp. 99–106.

[8] Lorido-Botran, T.; Miguel-Alonso, J.; Lozano, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, volume 12, no. 4, 2014: pp. 559–592.

[9] Iosup, A.; Li, H.; Jan, M.; et al. The grid workloads archive. *Future Generation Computer Systems*, volume 24, no. 7, 2008: pp. 672–686.

[10] Zukerman, M. Introduction to queueing theory and stochastic teletraffic models. *arXiv preprint arXiv:1307.2968*, 2013.

[11] Menasce, D. A.; Almeida, V. A.; Dowdy, L. W.; et al. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[12] Bolch, G.; Greiner, S.; de Meer, H.; et al. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[13] Danzig, P.; Mogul, J.; Paxson, V.; et al. The internet traffic archive. Lawrence Berkeley National Laboratory, 2008. URL `http://ita.ee.lbl.gov`

[14] Hellerstein, J. L.; Cirne, W.; Wilkes, J. Google cluster data. Google research blog, Jan, 2010. URL `https://github.com/google/cluster-data`

[15] Yang, L.; Foster, I.; Schopf, J. M. Homeostatic and tendency-based CPU load predictions. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, IEEE, 2003, pp. 9–pp.

[16] Wolski, R.; Spring, N. T.; Hayes, J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, volume 15, no. 5, 1999: pp. 757–768.

[17] Caron, E.; Desprez, F.; Muresan, A. Pattern matching based forecast of non-periodic repetitive behavior for cloud clients. *Journal of Grid Computing*, volume 9, no. 1, 2011: pp. 49–64.

[18] Herbst, N. R.; Huber, N.; Kounev, S.; et al. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and computation: practice and experience*, volume 26, no. 12, 2014: pp. 2053–2078.

[19] Huang, C.-J.; Guan, C.-T.; Chen, H.-M.; et al. An adaptive resource management scheme in cloud computing. *Engineering Applications of Artificial Intelligence*, volume 26, no. 1, 2013: pp. 382–389.

[20] Zhang, Q.; Zhani, M. F.; Zhang, S.; et al. Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th international conference on Autonomic computing*, ACM, 2012, pp. 145–154.

[21] Ali-Eldin, A.; Kihl, M.; Tordsson, J.; et al. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, ACM, 2012, pp. 31–40.

[22] Ardagna, D.; Casolari, S.; Colajanni, M.; et al. Dual time-scale distributed capacity allocation and load redirect algorithms for cloud systems. *Journal of Parallel and Distributed Computing*, volume 72, no. 6, 2012: pp. 796–808.

[23] Wang, X.; Du, Z.; Chen, Y. An adaptive model-free resource and power management approach for multi-tier cloud environments. *Journal of Systems and Software*, volume 85, no. 5, 2012: pp. 1135–1146.

[24] Kim, H.; Kim, W.; Kim, Y. A pattern-based prediction model for dynamic resource provisioning in cloud environment. *KSII Transactions on Internet and Information Systems (TIIS)*, volume 5, no. 10, 2011: pp. 1712–1732.

[25] Duy, T. V. T.; Yukinori, S.; Inoguchi, Y. A prediction-based green scheduler for datacenters in clouds. *IEICE TRANSACTIONS on Information and Systems*, volume 94, no. 9, 2011: pp. 1731–1741.

[26] Shen, Z.; Subbiah, S.; Gu, X.; et al. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ACM, 2011, p. 5.

[27] Santana, C.; Leite, J. C.; Mossé, D. Power management by load forecasting in web server clusters. *Cluster Computing*, volume 14, no. 4, 2011: pp. 471–481.

[28] Vasar, M.; Srirama, S. N.; Dumas, M. Framework for monitoring and testing web application scalability on the cloud. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, ACM, 2012, pp. 53–60.

[29] Iverson, M. A.; Ozguner, F.; Potter, L. C. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, IEEE, 1999, pp. 99–111.

[30] Dean, D. J.; Nguyen, H.; Gu, X. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*, ACM, 2012, pp. 191–200.

[31] Tan, Y.; Nguyen, H.; Shen, Z.; et al. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, IEEE, 2012, pp. 285–294.

[32] Li, H. Performance evaluation in grid computing: A modeling and prediction perspective. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, IEEE, 2007, pp. 869–874.

[33] Beloglazov, A.; Abawajy, J.; Buyya, R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, volume 28, no. 5, 2012: pp. 755–768.

[34] Jeyarani, R.; Nagaveni, N.; Ram, R. V. Design and implementation of adaptive power-aware virtual machine provisioner (APA-VMP) using swarm intelligence. *Future Generation Computer Systems*, volume 28, no. 5, 2012: pp. 811–821.

[35] Oliveira, D.; Ogasawara, E.; Ocaña, K.; et al. An adaptive parallel execution strategy for cloud-based scientific workflows. *Concurrency and Computation: Practice and Experience*, volume 24, no. 13, 2012: pp. 1531–1550.

[36] Zhao, W.; Peng, Y.; Xie, F.; et al. Modeling and simulation of cloud computing: A review. In *Cloud Computing Congress (APCloudCC), 2012 IEEE Asia Pacific*, IEEE, 2012, pp. 20–24.

[37] Sotiriadis, S.; Bessis, N.; Antonopoulos, N. Towards inter-cloud simulation performance analysis: Exploring service-oriented benchmarks of clouds in SimIC. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, IEEE, 2013, pp. 765–771.

[38] Altevogt, P.; Denzel, W.; Kiss, T. The IBM Performance Simulation Framework for Cloud. *IBM Research Technical Paper*. URL `http://domino.research.ibm.com/library/cyberdig.nsf/papers/041264DDC6C63D8485257BC800504EA2`

[39] Fittkau, F.; Frey, S.; Hasselbring, W. Cloud user-centric enhancements of the simulator cloudsim to improve cloud deployment option analysis. In *Service-Oriented and Cloud Computing*, Springer, 2012, pp. 200–207.

[40] Fonseca i Casas, P.; Casanovas, J. JGPSS, an open source GPSS framework to teach simulation. In *Winter Simulation Conference*, Winter Simulation Conference, 2009, pp. 256–267.

[41] Bazan, P.; Bolch, G.; German, R. WinPEPSY-QNS¿ Performance Evaluation and Prediction System for Queueing Networks. In *Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB), 2006 13th GI/ITG Conference*, VDE, 2006, pp. 1–4.

[42] Gunther, N. J. *Analyzing Computer System Performance with Perl:: PDQ*. Springer Science & Business Media, 2011.

[43] Bertoli, M.; Casale, G.; Serazzi, G. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, volume 36, no. 4, 2009: pp. 10–15, ISSN 0163-5999, doi: http://doi.acm.org/10.1145/1530873.1530877.

[44] Marzolla, M. The `qnetworks` Toolbox: A Software Package for Queueing Networks Analysis. In *Analytical and Stochastic Modeling Techniques and Applications, 17th International Conference, ASMTA 2010, Cardiff, UK, Proceedings*, *Lecture Notes in Computer Science*, volume 6148, edited by K. Al-Begain; D. Fiems; W. J. Knottenbelt, Springer, June14–16 2010, ISBN 978-3-642-13567-5, pp. 102–116.

[45] Hirel, C.; Tuffin, B.; Trivedi, K. S. Spnp: Stochastic petri nets. version 6.0. In *Computer Performance Evaluation. Modelling Techniques and Tools*, Springer, 2000, pp. 354–357.

[46] Hirel, C.; Sahner, R.; Zang, X.; et al. Reliability and performability modeling using SHARPE 2000. In *Computer Performance Evaluation. Modelling Techniques and Tools*, Springer, 2000, pp. 345–349.

[47] Bolch, G.; Herold, H. MOSEL–MOodeling Specification and Evaluation Language. 1995.

[48] Reussner, R.; Becker, S.; Koziolek, A.; et al. *Perspectives on the Future of Software Engineering*, chapter An Empirical Investigation of the Component-Based Performance Prediction Method Palladio. Springer Berlin Heidelberg, 2013, ISBN 978-3-642-37394-7, pp. 191–207, doi: 10.1007/978-3-642-37395-4_13. URL `http://dx.doi.org/10.1007/978-3-642-37395-4_13`

[49] Rolia, J.; Sevcik, K. The Method of Layers. *Software Engineering, IEEE Transactions on*, volume 21, no. 8, Aug 1995: pp. 689–700, ISSN 0098-5589, doi: 10.1109/32.403785.

[50] Perez, J.; Casale, G. Assessing SLA Compliance from Palladio Component Models. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, Sept 2013, pp. 409–416, doi: 10.1109/SYNASC.2013.60.

[51] Becker, M.; Becker, S.; Meyer, J. SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems. *Software Engineering*, volume 213, 2013: pp. 71–84.

[52] Rice, S. V.; Markowitz, H. M.; Marjanski, A.; et al. The SIMSCRIPT III programming language for modular object-oriented simulation. In *Proceedings of the 37th conference on Winter simulation*, Winter Simulation Conference, 2005, pp. 621–630.

[53] Schwetman, H. CSIM19: CSIM19: a powerful tool for building system models. In *Proceedings of the 33nd conference on Winter simulation*, IEEE Computer Society, 2001, pp. 250–255.

[54] Henderson, T. R.; Lacage, M.; Riley, G. F.; et al. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, volume 14, 2008.

[55] Varga, A.; et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM'2001)*, volume 9, sn, 2001, p. 65.

[56] Howell, F.; McNab, R. SimJava: A discrete event simulation library for java. *Simulation Series*, volume 30, 1998: pp. 51–56.

[57] Matloff, N. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, volume 2, 2008: p. 2009.

[58] Keogh, E. A decade of progress in indexing and mining large time series databases. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB Endowment, 2006, pp. 1268–1268.

[59] Babka, M. *Porovnání algoritm· predikce výroby elektrické energie fotovoltaické elektrárny*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2011.

[60] Brutlag, J. D. Aberrant Behavior Detection in Time Series for Network Monitoring. In *LISA*, volume 14, 2000, pp. 139–146.

[61] Kalekar, P. S. Time series forecasting using holt-winters exponential smoothing. *Kanwal Rekhi School of Information Technology*, volume 4329008, 2004: pp. 1–13.

[62] Hyndman, R. J. Forecast estimation, evaluation and transformation. Hyndsight blog, 2010, [Online; downloaded 4-February-2017]. URL `http://robjhyndman.com/hyndsight/forecastmse/`

[63] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL `https://www.R-project.org`

[64] Hyndman, R. J. *forecast: Forecasting functions for time series and linear models*. 2011, r package version 4.0. URL `http://github.com/robjhyndman/forecast`

[65] Lundholm, M. *Introduction to R's time series facilities*. 2011, ver. 1.3. URL `http://people.su.se/~lundh/reproduce/introduction_ts.pdf`

[66] Hyndman, R. J. CRAN Task View: Time Series Analysis. 2013, version 10 March 2013. URL `http://cran.r-project.org/web/views/TimeSeries.html`

[67] McLeod, A. I.; Yu, H.; Mahdi, E. Time series analysis with R. *Handbook of statistics*, volume 30, 2011: pp. 78–93.

[68] Hyndman, R. J.; Athanasopoulos, G. *Forecasting: principles and practice*. OTexts, 2012. URL `http://otexts.com/fpp/8/9/`

[69] Crone, S. Forecasting with Artificial Neural Networks. Tutorial at the 2005 IEEE Summer School in Computational Intelligence EVIC'05, Santiago, Chile, 2005. URL `http://www.neural-forecasting.com/tutorials.htm`

[70] Croarkin, C.; Tobias, P.; Filliben, J.; et al. *NIST/SEMATECH e-handbook of statistical methods*. NIST/SEMATECH, 2003, version 1-April-2012. URL `http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4.htm`

[71] Vondra, T.; Šedivý, J.; Castro, J. Modifying CloudSim to Accurately Simulate Interactive Services for Cloud Autoscaling. *Concurrency and Computation - Practice and Experience*, volume 28, no. 9, 2016: pp. 0–0.

[72] Wickremasinghe, B.; Calheiros, R. N.; Buyya, R. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, IEEE, 2010, pp. 446–452.

[73] Bessis, N.; Sotiriadis, S.; Xhafa, F.; et al. Cloud scheduling optimization: a reactive model to enable dynamic deployment of virtual machines instantiations. *Informatica*, volume 24, no. 3, 2013: pp. 357–380.

[74] Class Sim_stat, SimJava v.2.0 documentation. University of Edinburgh, 2002, [Online; downloaded 28-September-2015]. URL `http://www.dcs.ed.ac.uk/home/simjava/doc/eduni/simjava/Sim_stat.html`

[75] R Core Team. *R Data Import/Export*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL `https://www.R-project.org`

[76] Grothendieck, G. Time series in half hourly intervals- how do i do it? R-SIG-Finance news group, 2010, [Online; downloaded 4-February-2017]. URL `https://stat.ethz.ch/pipermail/r-sig-finance/2010q3/006729.html`

[77] Coghlan, A. A Little Book of R for Time Series. 2010, [Online; downloaded 12-March-2013]. URL `http://a-little-book-of-r-for-time-series.readthedocs.org/en/latest/`

[78] Hyndman, R. J. Cyclic and seasonal time series. Hyndsight blog, 2011, [Online; downloaded 4-February-2017]. URL `http://robjhyndman.com/hyndsight/cyclicts/`

[79] Nielsen, H. B. Non-stationary time series and unit root testing. Lecture for Econometrics II, Department of Economics, University of Copenhagen, 2005, [Online; downloaded 4-February-2017]. URL `http://www.econ.ku.dk/metrics/Econometrics2_05_II/Slides/08_unitroottests_2pp.pdf`

[80] Bhattach, K.; Stigler, M.; Frain, J. Which one is better? Discussion in RMetrics, 2010, [Online; downloaded 4-February-2017]. URL `http://r.789695.n4.nabble.com/Which-one-is-better-td991742.html`

[81] Nau, R. Seasonal ARIMA models. Course notes for Decision 411 Forecasting, Fuqua School of Business, Duke University, 2005, [Online; downloaded 4-February-2017]. URL `http://people.duke.edu/~rnau/seasarim.htm`

[82] STAT 510 - Applied Time Series Analysis, Chapter 4: Seasonal Models. Online course at Department of Statistics, Eberly College of Science, Pennsylvania State University, 2013, [Online; downloaded 4-February-2017]. URL `https://onlinecourses.science.psu.edu/stat510/?q=book/export/html/50`

[83] Hyndman, R. J. Forecasting with long seasonal periods. Hyndsight blog, 2010, [Online; downloaded 4-February-2017]. URL `http://robjhyndman.com/hyndsight/longseasonality/`

[84] R Development Core Team. *arima0: ARIMA Modelling of Time Series*. R Foundation for Statistical Computing, Vienna, Austria, 2010, preliminary version. URL `http://stat.ethz.ch/R-manual/R-patched/library/stats/html/arima0.html`

[85] Sevcik, P. Defining the application performance index. *Business Communications Review*, volume 20, 2005.

[86] Canadilla, P. *queueing: Analysis of Queueing Networks and Models*. 2015, r package version 0.2.6. URL `http://CRAN.R-project.org/package=queueing`

[87] Spinner, S.; Kounev, S.; Zhu, X.; et al. Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation. In *Self-Adaptive and Self-Organizing Systems (SASO), 2014 IEEE Eighth International Conference on*, IEEE, 2014, pp. 157–166.

[88] Vondra, T. A Queue System for Batch Scientific Computations on Private IaaS Clouds. In *POSTER 2012: 16th International Student Conference on Electrical Engineering*, Prague: CTU, 2012, p. IC04.

[89] Vondra, T.; Sedivy, J. Maximizing utilization in private iaas clouds with heterogenous load. In *CLOUD COMPUTING, The Third International Conference on Cloud Computing, GRIDs, and Virtualization. Nice, France*, IARIA, 2012, pp. 169–173.

[90] Šín, J. *Production Control Optimization in SaaS*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2011.

[91] Danko, K. *Automatic scaling in private IaaS*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2013.

[92] Pulec, P. *Srovnání open-source systémů cloudů PaaS*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2014.

[93] Tkadleček, J. *Plánovací algoritmy ve frontách úloh*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2014.

[94] Uhrín, M. *Data movement in hybrid clouds*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2014.

[95] Kábrt, T. *Automatické nasazení aplikací v cloudu řízené modelem výkonnosti*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2015.

[96] Židek, M. *Monitorování privátního cloudu*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2015.

[97] Špak, M. *Autoscaling v Cloud Foundry*. Bachelors thesis, Czech Technical University, Faculty of Information Technology, 2016.

[98] Ferdman, M.; Adileh, A.; Kocberber, O.; et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-0759-8, pp. 37–48, doi: 10.1145/2150976.2150982. URL `http://doi.acm.org/10.1145/2150976.2150982`

[99] Urgaonkar, B.; Shenoy, P.; Chandra, A.; et al. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, volume 3, no. 1, 2008: p. 1.

[100] Vondra, T.; Šedivỳ, J. Cloud autoscaling simulation based on queueing network model. *Simulation Modelling Practice and Theory*, volume 70, 2017: pp. 83–100.

[101] SPENASSATO, D.; Andréa, C. T.; BORNIA, A. C.; et al. Dow Jones Sustainability Index: Use of forecasting models to assist decision making. *Espacios*, volume 36, no. 11, 2015.

[102] Jia, C.; Wei, L.; Wang, H.; et al. Study of track irregularity time series calibration and variation pattern at unit section. *Computational intelligence and neuroscience*, volume 2014, 2014: p. 34.

[103] Kouba, Z. *Optimalizace sít'ové architektury pro Hadoop Mapreduce*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2015.

[104] Vondra, T. *Load Forecasting for Cloud Computing*. Minimal phd progress report, Czech Technical University, Faculty of Electrical Engineering, 2013.

[105] Vondra, T.; Sedivy, J. *Od hostingu ke cloudu*. CTU, Faculty of Electrical Engineering, Gerstner Laboratory, 2011, Research Report GL 229/11.

[106] VONDRA, T.; MICHALIČKA, P.; ŠEDIVÝ, J. UpCF: Automatic Deployment of PHP Applications to Cloud Foundry PaaS. In *Mezinárodní Masarykova konference pro doktorandy a mladé vědecké pracovníky, Hradec Králové*, Magnanimitas, 2013, pp. 3982–3991.

[107] Vondra, T.; Sedivy, J. *Cloudy Typu Platform as a Service*. CTU, Faculty of Electrical Engineering, Gerstner Laboratory, 2011, Research Report GL 230/11.

[108] Drdlíček, M. *Aplikace pro mobilní telefony a chytré televizory pro neziskovou TV*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2016.

[109] Paločko, V. *Mobilní Android aplikace pro turistický portál*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2015.

[110] Věžník, D. *Interaktivní HTML5 video přehrávač s playlistem*. Bachelors thesis, Czech Technical University, Faculty of Information Technology, 2015.

[111] Luňák, M. *Webová prezentace s HTML5 streamingem pro neziskovou TV*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2014.

[112] Motyčka, J. *SaaS Autoservis*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2014.

[113] Sukhotin, M. *Mobile iOS application for a non-profit TV*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2014.

[114] Michalička, P. *Automatic Deployment to PaaS Cloud*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2012.

[115] Vydržel, L. *Emulátor plné chytré baterie*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2012.

Appendix A

List of Publications and Grants

List of candidate's work related to the thesis

The percentage of contribution is even for all listed authors of each publication, i.e. 50%-50% in the case of two authors and 33%-33%-33% in the case of three authors.

Journals (Impact)

- Vondra, T.; Šedivỳ, J. Cloud autoscaling simulation based on queueing network model. *Simulation Modelling Practice and Theory*, volume 70, 2017: pp. 83–100

- Vondra, T.; Šedivý, J.; Castro, J. Modifying CloudSim to Accurately Simulate Interactive Services for Cloud Autoscaling. *Concurrency and Computation - Practice and Experience*, volume 28, no. 9, 2016: pp. 0–0

Journals (Reviewed)

- Vondra, T.; Šedivý, J. Maximizing Utilization in Private IaaS Clouds with Heterogenous Load through Time Series Forecasting. *Int. J. On Advances in Syst. and Measurements*, volume 6, no. 1 and 2, 2013: pp. 149–165

  The paper has been cited in:

  - Weingärtner, R.; Bräscher, G. B.; Westphall, C. B. Cloud resource management: A survey on forecasting and profiling models. *Journal of Network and Computer Applications*, volume 47, 2015: pp. 99–106
  - SPENASSATO, D.; Andréa, C. T.; BORNIA, A. C.; et al. Dow Jones Sustainability Index: Use of forecasting models to assist decision making. *Espacios*, volume 36, no. 11, 2015

  - Jia, C.; Wei, L.; Wang, H.; et al. Study of track irregularity time series calibration and variation pattern at unit section. *Computational intelligence and neuroscience*, volume 2014, 2014: p. 34

Conferences

- Vondra, T.; Sedivy, J. Maximizing utilization in private iaas clouds with heterogenous load. In *CLOUD COMPUTING, The Third International Conference on Cloud Computing, GRIDs, and Virtualization. Nice, France*, IARIA, 2012, pp. 169–173

  The paper has been cited in:

  - Weingärtner, R.; Bräscher, G. B.; Westphall, C. B. Cloud resource management: A survey on forecasting and profiling models. *Journal of Network and Computer Applications*, volume 47, 2015: pp. 99–106
  - Kouba, Z. *Optimalizace sít'ové architektury pro Hadoop Mapreduce*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2015

- Vondra, T. A Queue System for Batch Scientific Computations on Private IaaS Clouds. In *POSTER 2012: 16th International Student Conference on Electrical Engineering*, Prague: CTU, 2012, p. IC04

Research Reports

- Vondra, T. *Load Forecasting for Cloud Computing*. Minimal phd progress report, Czech Technical University, Faculty of Electrical Engineering, 2013

- Vondra, T.; Sedivy, J. *Od hostingu ke cloudu*. CTU, Faculty of Electrical Engineering, Gerstner Laboratory, 2011, Research Report GL 229/11

Grants (as proposer)

- Vondra, T.: Grant Agency of the Czech Technical University in Prague, grant no. SGS13/141/OHK3/2T/13, Application of artificial intelligence methods to cloud computing problems

Master's and Bachelor's theses (as supervisor or consultant)

- Špak, M. *Autoscaling v Cloud Foundry*. Bachelors thesis, Czech Technical University, Faculty of Information Technology, 2016

- Židek, M. *Monitorování privátního cloudu*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2015

- Kábrt, T. *Automatické nasazení aplikací v cloudu řízené modelem výkonnosti*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2015

- Uhrín, M. *Data movement in hybrid clouds*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2014

- Tkadleček, J. *Plánovací algoritmy ve frontách úloh*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2014

- Pulec, P. *Srovnání open-source systémů cloudů PaaS*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2014

- Klepac, M. *Private IaaS cloud comparison*. Bachelors thesis, Czech Technical University, Faculty of Information Technology, 2013

- Danko, K. *Automatic scaling in private IaaS*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2013

- Šín, J. *Production Control Optimization in SaaS*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2011

List of candidate's work non-related to the thesis

Conferences

- VONDRA, T.; MICHALIČKA, P.; ŠEDIVÝ, J. UpCF: Automatic Deployment of PHP Applications to Cloud Foundry PaaS. In *Mezinárodní Masarykova konference pro doktorandy a mladé vědecké pracovníky, Hradec Králové*, Magnanimitas, 2013, pp. 3982–3991

Research Reports

- Vondra, T.; Sedivy, J. *Cloudy Typu Platform as a Service*. CTU, Faculty of Electrical Engineering, Gerstner Laboratory, 2011, Research Report GL 230/11

Grants (as team member)

- Gogár, T.: Internal grant of the dept. of Cybernetics, 2015, Focused web crawling

- Šedivý, J.: Grant Agency of the Czech Technical University in Prague, grant no. SGS11/128/OHK3/2T/13, Rapid Application Development Tools and Intelligent Control in Cloud Computing

Master's and Bachelor's theses (as supervisor or consultant)

- Drdlíček, M. *Aplikace pro mobilní telefony a chytré televizory pro neziskovou TV*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2016
- Paločko, V. *Mobilní Android aplikace pro turistický portál*. Master's thesis, Czech Technical University, Faculty of Information Technology, 2015
- Věžník, D. *Interaktivní HTML5 video přehrávač s playlistem*. Bachelors thesis, Czech Technical University, Faculty of Information Technology, 2015
- Luňák, M. *Webová prezentace s HTML5 streamingem pro neziskovou TV*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2014
- Motyčka, J. *SaaS Autoservis*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2014
- Sukhotin, M. *Mobile iOS application for a non-profit TV*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2014
- Michalička, P. *Automatic Deployment to PaaS Cloud*. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, 2012
- Vydržel, L. *Emulátor plné chytré baterie*. Bachelors thesis, Czech Technical University, Faculty of Electrical Engineering, 2012